



GRDC: A COLLABORATIVE FRAMEWORK FOR RADIOLOGICAL BACKGROUND  
AND CONTEXTUAL DATA ANALYSIS  
Project: LB11-SAM\_Feasibility\_Study-PD3SJ

Brian J. Quiter, Lavanya Ramakrishnan, and Mark S. Bandstra  
Lawrence Berkeley National Laboratory

Submitted to the  
Office of Defense Nuclear Nonproliferation Research and Development  
National Nuclear Security Administration

December 2015

Report Number: LBNL-XXXXXXX

## DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or The Regents of the University of California.

## COPYRIGHT NOTICE

This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

## ACKNOWLEDGMENTS

This work was supported by the National Nuclear Security Administration (NNSA), Office of Defense Nuclear Nonproliferation Research and Development, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

## SUMMARY

The Radiation Mobile Analysis Platform (RadMAP) is unique in its capability to collect both high quality radiological data from both gamma-ray detectors and fast neutron detectors and a broad array of contextual data that includes positioning and stance data, high-resolution 3D radiological data from weather sensors, LiDAR, and visual and hyperspectral cameras. The datasets obtained from RadMAP are both voluminous and complex and require analyses from highly diverse communities within both the national laboratory and academic communities. Maintaining a high level of transparency will enable analysis products to further enrich the RadMAP dataset. It is in this spirit of open and collaborative data that the RadMAP team proposed to collect, calibrate, and make available online data from the RadMAP system. The Berkeley Data Cloud (BDC) is a cloud-based data management framework that enables web-based data browsing visualization, and connects curated datasets to custom workflows such that analysis products can be managed and disseminated while maintaining user access rights. BDC enables cloud-based analyses of large datasets in a manner that simulates real-time data collection, such that BDC can be used to test algorithm performance on real and source-injected datasets. Using the BDC framework, a subset of the RadMAP datasets have been disseminated via the Gamma Ray Data Cloud (GRDC, <https://grdc.nersc.gov>) that is hosted through the National Energy Research Science Computing (NERSC) Center, enabling data access to over 40 users at 10 institutions.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>BDC Organization</b>	<b>3</b>
2.1	Front end – Drupal web interface . . . . .	3
2.2	API connections & Source Injection . . . . .	9
2.3	Back end Infrastructure . . . . .	11
2.3.1	Data Ingest . . . . .	11
2.3.2	Data Management and Extraction . . . . .	12
2.3.2.1	Platform and Run Configuration Management . . . . .	13
2.3.2.2	Sensor data organization and retrieval . . . . .	15
2.3.2.3	Process/workflow management . . . . .	18
2.3.2.4	User ownership and permissions . . . . .	19
2.3.3	Workflow Management . . . . .	20
2.3.4	Web Services . . . . .	21
2.4	Interaction with frontend . . . . .	21
<b>3</b>	<b>GRDC Status</b>	<b>23</b>
3.1	Capabilities . . . . .	23
3.2	Data Available on GRDC . . . . .	23
<b>4</b>	<b>Outlook</b>	<b>24</b>
<b>A</b>	<b>Appendix A - Glossary</b>	<b>27</b>
<b>B</b>	<b>Appendix B - BDC User Manual</b>	<b>28</b>

# GRDC: A COLLABORATIVE FRAMEWORK FOR RADIOLOGICAL BACKGROUND AND CONTEXTUAL DATA ANALYSIS

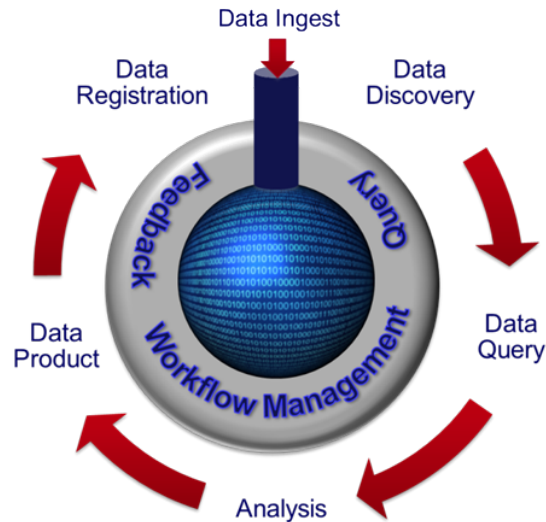
## 1 Introduction

Many nuclear security radiation detection applications make wide use of gamma radiation sensing to detect, locate, and characterize nuclear and radiological material. When these activities occur in uncontrolled environments the complexity and variability of the benign background can confound detection algorithms and methods, worsening sensitivity or causing false alarms - especially for large area/low resolution sensor systems[1]-[5]. The loss of effectiveness of such detector systems due to environmental complexities can, however, be mitigated by algorithms that better account for the environment[6]. Lawrence Berkeley National Laboratory and the University of California, Berkeley have developed and are fielding the Radiological Multisensor Analysis Platform (RadMAP), which is a mobile, truck-based detection platform that acquires background radiation data synchronously with a suite of contextual sensors. The RadMAP effort has several goals including: understanding how the environment influences radioactive backgrounds; enabling the development of algorithms that leverage contextual sensors to improve sensitivity; identifying which contextual sensors provide value to the radiological detection problem; and obtaining valuable datasets that can be used by the RadMAP team and external collaborators to develop and test advanced detection methods, benchmark and validate modeling methods, and to test other nuclear security-related analyses that may be independent of radiation detection. To facilitate the goal of ensuring that the obtained datasets are valuable and to enable our own analyses, the RadMAP team required a data management system that could support data visualization, data dissemination, and the ability to connect algorithms directly to the data such that voluminous data could be analyzed remotely.

The ongoing evolution of the RadMAP system caused additional requirements for BDC. The RadMAP system has undergone sensor upgrades, hardware synchronization upgrades and improvements/refinements of sensor calibrations throughout the project. As such, data, data calibrations, and dataset availability have evolved over time. Through the implementation of a relational database (RDB), software for managing data and the RDB, file descriptors and file descriptor libraries, tables that describe sensors, platforms, data versions and data collection configurations, the BDC has been developed to manage many complexities that arise when collecting disparate data streams generated from continually-evolving and long-running projects.

The goal of the BDC software package is to manage and disseminate disparate and complex datasets such that a wider set of researchers can access and understand the data. This can enable a collaborative environment wherein the strengths of various research groups can be applied to measured data, thereby maximizing its value. The environment can also facilitate side-by-side comparisons of data collection hardware and analysis methods such that strengths and weaknesses can be easily and transparently identified. As the BDC software package matures, it is expected that the ability to manage and visualize large datasets will improve capabilities in a broad range of applied and basic sciences; from the managing Airborne Measuring System (AMS) data and improving reachback capabilities to identifying nuclear data shortcomings via analysis of benchmarking and proof-of-principle experiments and models.

This report describes the organization of the BDC software package in Section 2, highlighting functionality built into the web-based frontend and the RDB. Section 3 summarizes the state of the BDC code package and the data disseminated and managed using GRDC. Section 4 highlights strengths and shortcomings within the existing BDC code base and suggests potential developments that would further augment the utility of this unique data management capability.



**Figure 1.** BDC concept: data management enabling data queries, analyses and dataproduct registration, resulting in the potential for the data quality and applications to feedback into planning additional measurements and analyses.

## 2 BDC Organization

The GRDC framework is based on a software platform, the Berkeley Data Cloud (BDC) that was developed to both disseminate RadMAP data as well as to enable data replay for the ARES advanced technology demonstration (ATD). The following describes the organization of the BDC from the user's perspective in Sections 2.1 and 2.2, and describes the more technical organization of BDC in Sections 2.3 and 2.3.2.

### 2.1 Front end – Drupal web interface

The primary means of browsing data and initiating data queries is through a front end website based in Drupal, an open source Content Management System[7]. All BDC instances have been protected by user logins. Upon logging in, a user is taken to the front page of the site. This is primarily a place for news updates and serves to hold links to user instructions. Any logged on user has the capability to create blog entries which could serve as a forum to publicize their work. All data on a BDC website are accessed within the WORKFLOW tab. Once there, a user may populate their Workspace by selecting one or more Runs from the Workspace Settings Dialog Box. In the Workspace Settings Dialog Box, filters may optionally be applied to help identify a Run of interest. At least one Run must be selected to visualize and query.

With a Workspace populated by one or more Runs, the user selects Runs whose data they want to visualize and/or query and then select Load data into browser. At this point, a visualization map shows tracks corresponding to selected Runs. Filters may be optionally applied either using the spatial filter box or by creating a 1-dimensional Histogram tool and selecting a portion of the histogram data – or both. The spatial data can also be colored using the Map Dimension tool and selecting any 1-dimensional Data Product by which to color.

The map interface uses the JavaScript library, polymaps.js to display the data over Bing satellite map images[8][9]. It provides the usual functionality of panning and zooming, identical to Bing or Google Maps.



Figure 2. Front Page of the GRDC website.

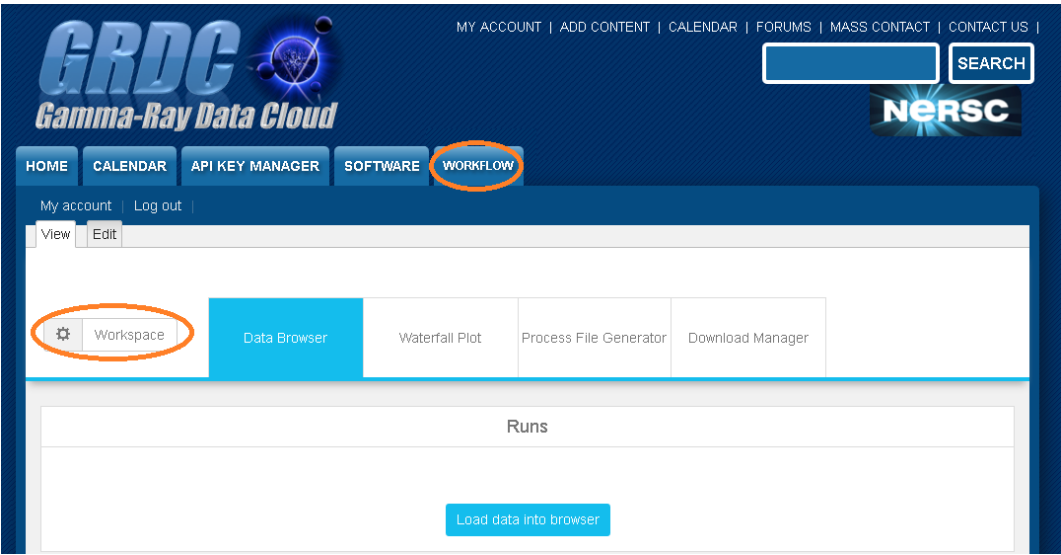
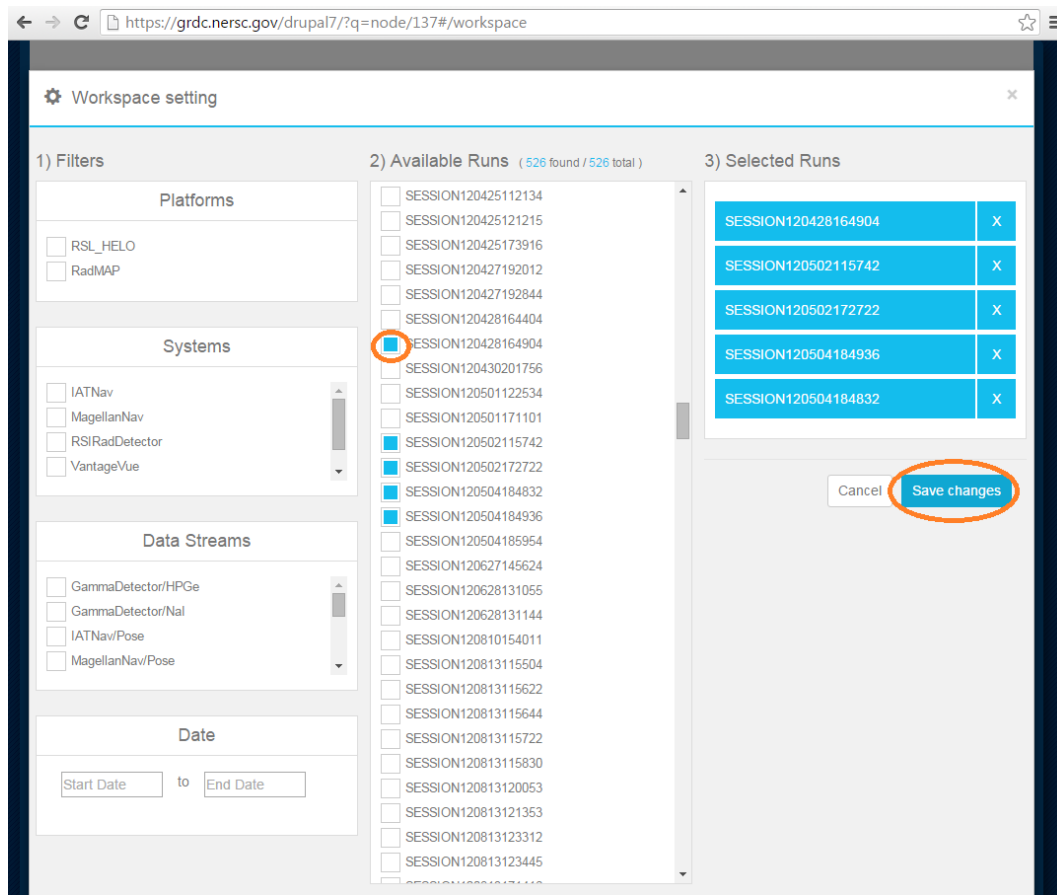


Figure 3. Empty GRDC workspace. Users must select the dialog box to populate the workspace.

In order to reflect which data are available given the different selections, the map and histograms respond in real time to any filter selections that are made by the user using the JavaScript library crossfilter.js[10]. For example, selecting a geographic region on the map in the figure below also updates the Timestamps histogram to show only the timestamps from within that geographic region. The histogram charts are drawn using the D3.js library[11].

To retrieve the data loaded in the Browser for download, the user must select Query below the visualization window, select at least one Data Product, and press the Submit query button. This will spawn a



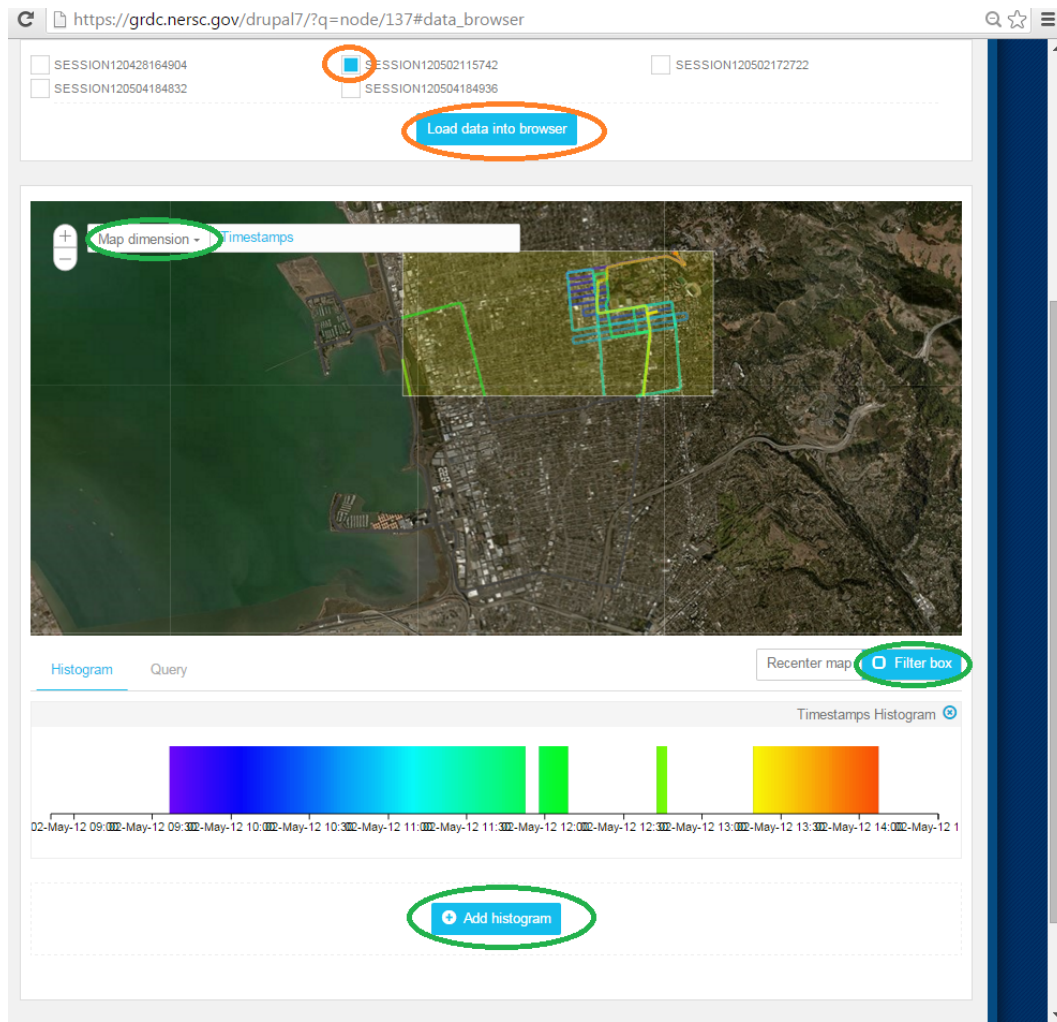


**Figure 4.** GRDC workspace dialog box.

Database Query (see Section 2.3.2.2, and will populate a row in the Download Manager table, which was custom-designed for BDC using jQuery libraries[12], a JavaScript library package that facilitates event handling (like the creation of a Query) and the manipulation of hyper-text markup language (HTML)-based web-pages. Initially, the status shown for this Query will be In Progress(0% Done). After some time, this will be changed to Available, at which point, clicking on Available will initiate download of the HDF5 [13] containing the requested data. HDF5 is a scientific data model, library, and file format for storing and managing data that is designed for flexible and efficient I/O and for high volume and complex data.

The Drupal website has other features that facilitate data access. Beyond direct download, BDC enables users to connect to data via an Application Programming Interface (API). This will be described in further detail in Section 2.2, but the website enables the user to generate unique random keys that provide the security for API connections. Keys have expiration dates that can be configured.

Lastly, the BDC websites have two capabilities that were specifically generated for the ARES program, but have broader functionality and utility. The first is the capability to simulate a remote API connection by downloading a Process File, that provides the BDC libraries with the data and type casting information necessary to ensure that a user-created processing pipeline can interact and upload dataproducts to the BDC using the remote connection. Processes are parts of a Workflow. At present a BDC developer must create a Workflow and processes for a user. At that point, the Process File Generator tab within the Drupal website will contain within the Select a Workflow drop-down menu all the workflows available to a user. By selecting a workflow, the Processes box will be populated with the processes associated with the selected Workflow. Selecting a processes will populate the inputs and outputs (I/O) boxes with all

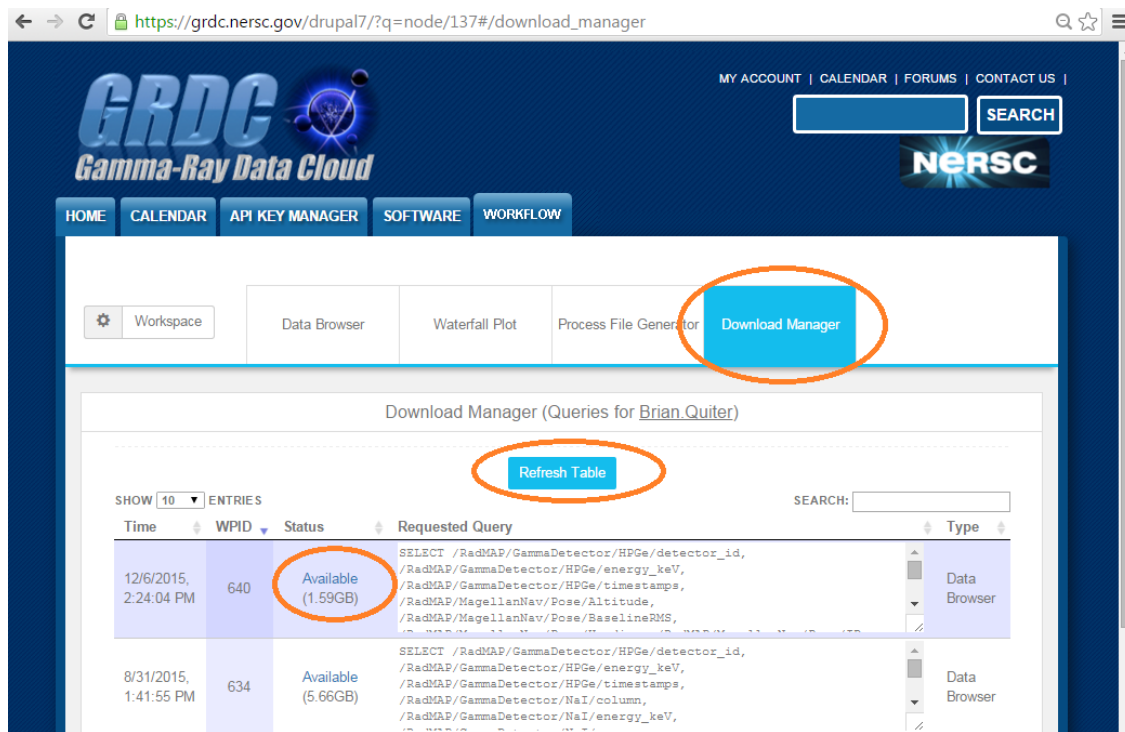


**Figure 5.** Figure showing a selected run that has been loaded into the GRDC polymaps display application and has had filters applied using crossfilter and D3. The required steps for data retrieval are highlighted as orange ovals, whereas the optional queries are shown in green. The selection of the Query tab has yet to occur in this image

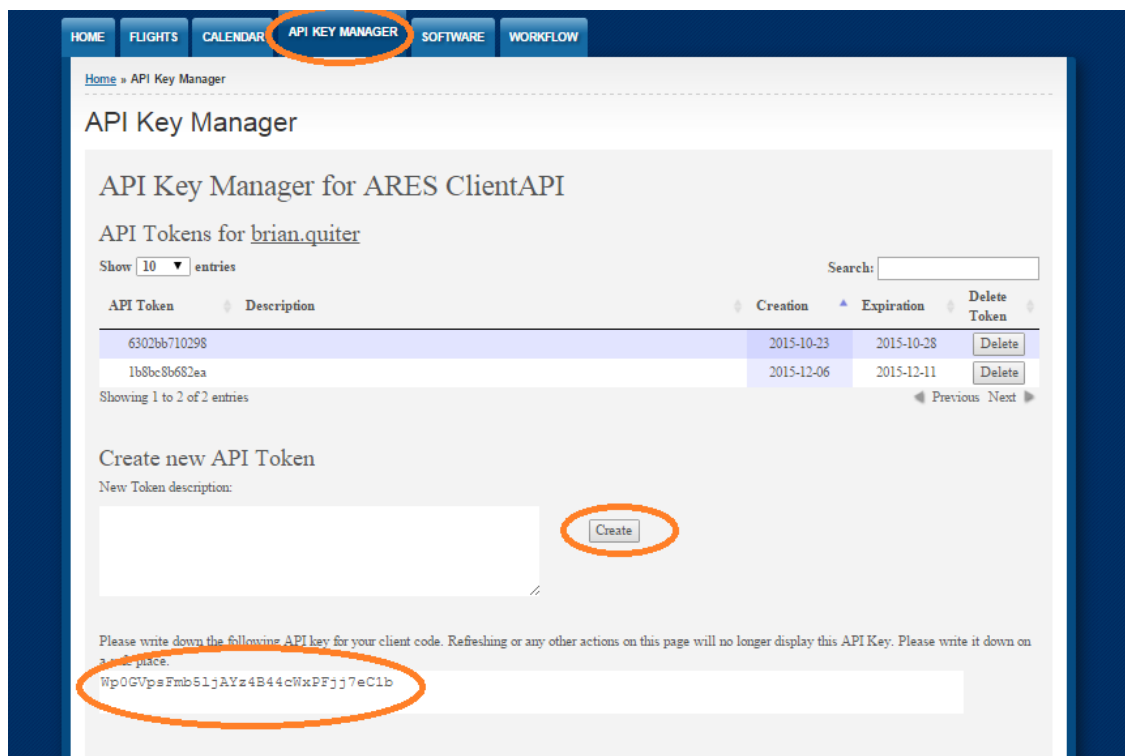
pre-defined I/O data streams. The at least one Run containing the proper inputs must be loaded in the Workspace in order for this to function. After selecting a Process, the user may select the Run for which the Process File is desired and press the Download Process File button, which populates a new entry in the Download Manager.

The second capability that is presently unique to the ARES implementation of the BDC is the Source Injection (SI) interface. On the Source Inject tab, a user selects a Run within their Workspace and presses the Set Up Source Injection Parameters button.

That will populate a browser-like graphic that indicates the portion of the Run the source injection calculation will take place, which are varied by modifying the Start and End times. The position of the source set either by entering values within the Latitude, Longitude and Altitude boxes or by clicking a point on the browser map. The type and activity of the injection source and the database file from which the injection will be selected are additionally specified, along with some parameters that define the numeric precision of the source injection calculation (along with default settings). Thereafter, the user selects the Generate Parameter File button, which produces a JavaScript Object Notation (JSON[14]) source injection parameter file that is user-specified upon connection via API to the BDC. The source injection parameter file could easily be produced or modified outside of the web GUI, enabling source



**Figure 6.** The GRDC download manager.



**Figure 7.** The BDC API Key manager/generator. The generated key (circled) disappears from the box after navigation away from page.

injection to generate moving source by discretizing the path as a series of static source positions and associated time ranges.

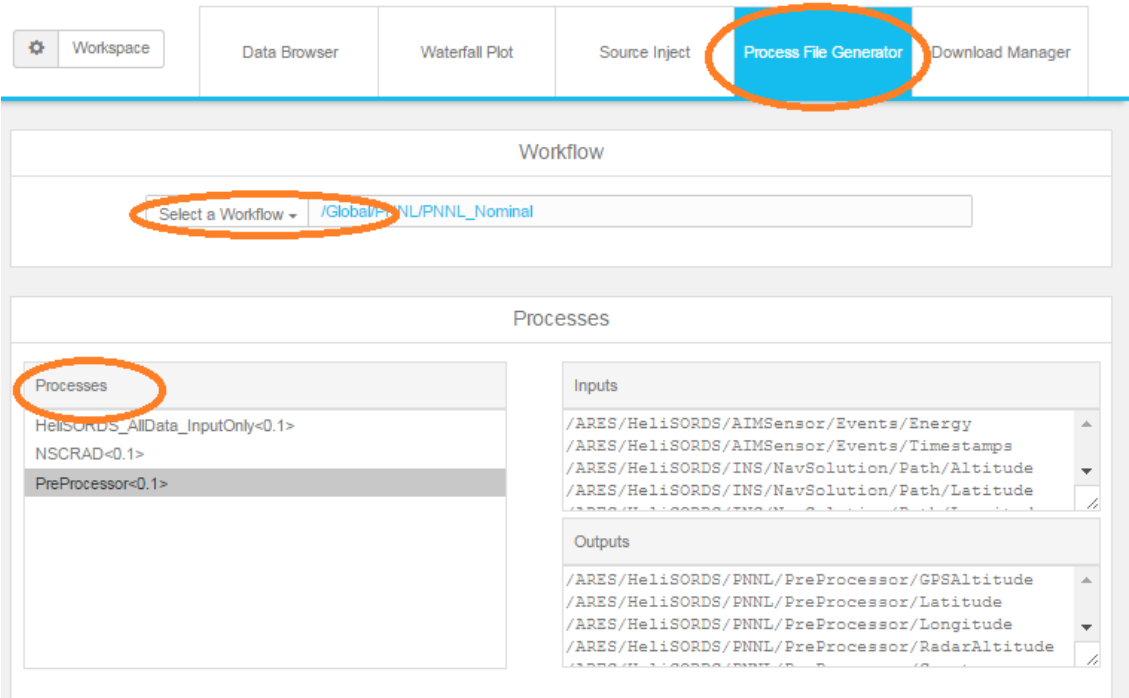


Figure 8. The ARES Process File Generator page that generated datafiles used in the connect\_local\_process API connection.

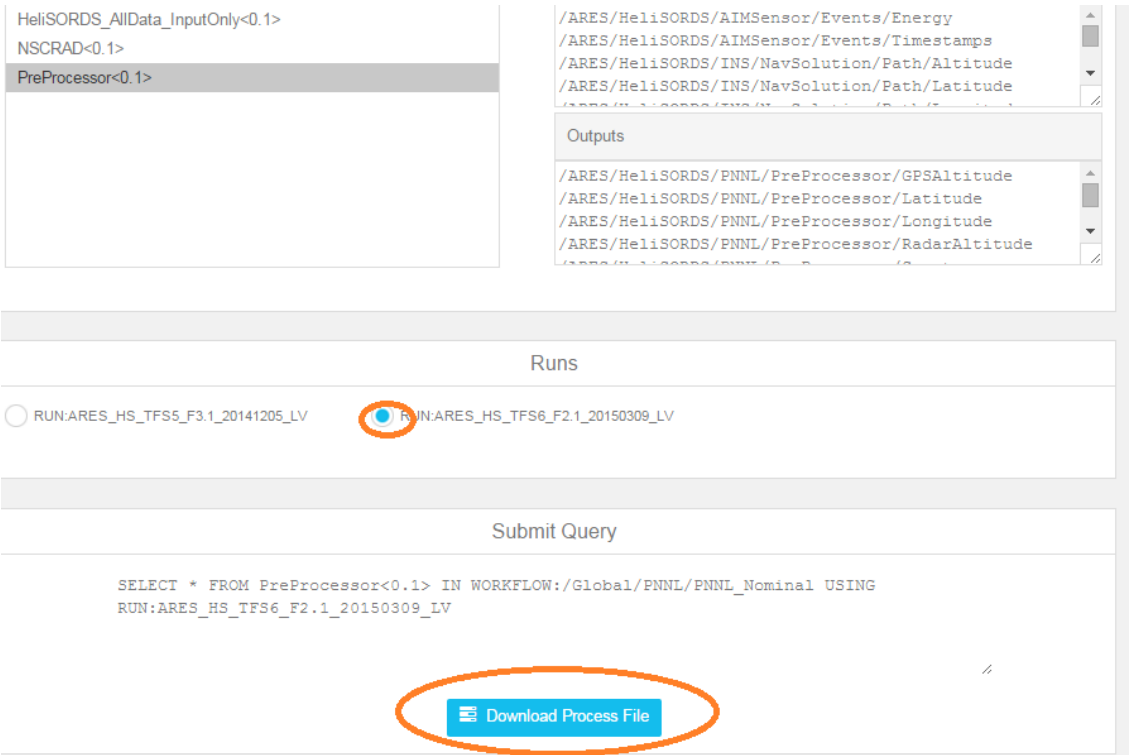
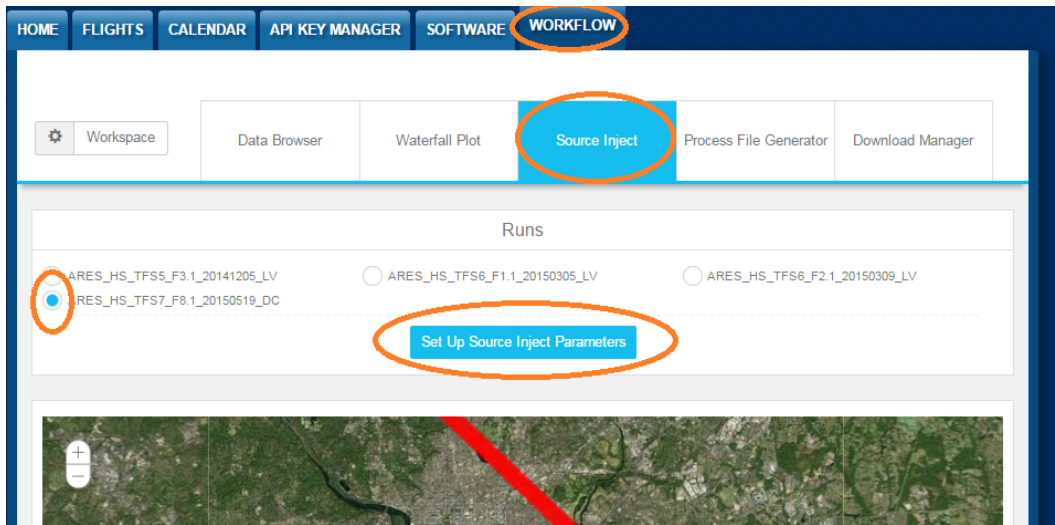
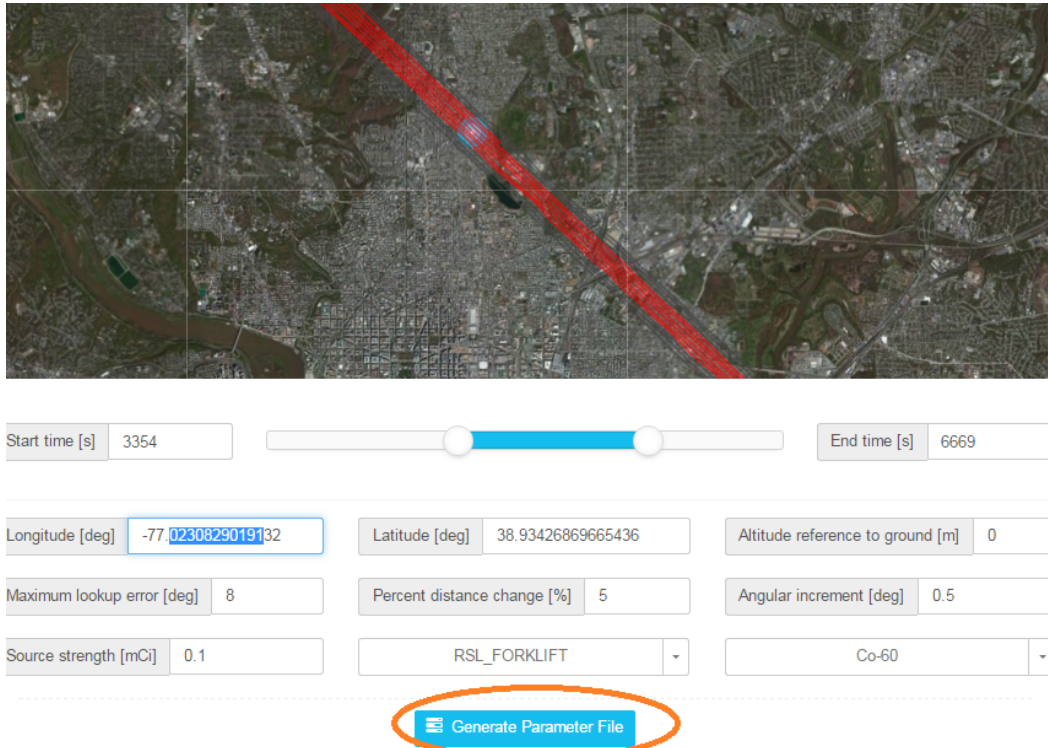


Figure 9. The ARES Process File Generator page with a Run selected and ready to download the file.



**Figure 10.** The ARES Source Inject page, loading a Run into the page and interface.



**Figure 11.** The ARES Source Inject page, selecting parameters and generating the JSON parameter file.

## 2.2 API connections & Source Injection

Algorithms can most efficiently access data stored within a BDC instance by connection via the client API. The client API is invoked by including BDC libraries within the source code of an algorithm. Libraries are available for software based in C/C++, java, Python, and Matlab on Linux, Windows, and Mac OSX, as well as C# on Windows only. There are three methods of API connection that are listed in increasing complexity: `connect_local`, `connect_local_process`, and `connect_remote`. The reasons to select one connection connection method versus another, as well as high-level functionality are first described, followed by a presentation of the functionalities – as provided by methods contained within the



API libraries. One method is the capability to access datasets for source injection, which was developed specifically for the ARES ATD, yet could provide appreciable value for users of all BDC instances.

Using `connect.local`, a user can locally stream a downloaded BDC HDF5 datafile into their algorithm without any interaction with the BDC data server beyond the initial download (or independent creation) of an HDF5 file that conforms to the `BdcHdf5Accessor` format (see Sec. 2.3), such as one retrieved from the web interface. This method was implemented to facilitate rapid prototyping, but outputs from `connect.local` processes are unknown to the data service. As such they are not managed in the BDC database.

The `connect.local.process` and `connect.remote` functionalities were developed to enable a user to more directly interface with the BDC and to simulate real-time operation. The `connect.local.process` method can be considered an intermediate step between `connect.local` and `connect.remote`. Before executing an algorithm containing either `connect.local.process` or `connect.remote` functionalities, the process must first be registered within the BDC. The act of registering a process defines the input that must be passed to the process and the outputs, if any, that will be generated by the process and their respective formats. Additionally, the parent processes of the inputs must be defined to enable the data service to ensure that all required inputs are available based on whether parent processes have been completed. Only processes for which all input parent processes are completed can be executed. Database considerations for process management are discussed in Section 2.3.2.3. An API connection using `connect.local.process` involves downloading a process file from a BDC webpage. The process file is an HDF5 file that contains all necessary input data for the specified process, the information describing the process inputs and output, and the time limits associated the data. When a process file is available within the client code environment, the `connect.local.process` may be invoked where the input data names, the chunking time, and the input process file name are passed as arguments. The connection handle object and the number of chunks available are returned. Chunking is used to describe the incremental passage of data between the BDC and a client code. A chunk corresponds to a time increment in which all input datastreams that have data elements within the time range will be passed to the client code. A client algorithm may then loop over the number of available chunks (or fewer), call `get_next_chunk` to receive the next increment of requested datastreams, perform analyses, then *stage* the registered process outputs and subsequently *publish* them. Staging locally establishes the data packet to be returned to the BDC and *publish* is the call that transmits the data. When the connection is due to a `connect.local.process` method, *publish* has no effect, but *stage* still ensures that the outputs have been properly formatted.

The `connect.remote` method contains all the functionality described for the `connect.local.process`, but is invoked to perform an internet connection using JSON and BSON (the binary equivalent of JSON)[15] to pass data. There is no need to download a process file, prior to connection, and the specific web address to which the connection shall be made must be specified along with the user name and a valid API key. Thereafter client algorithm functionality is identical to the `connect.local.process`, but the published output data is uploaded to the BDC server (again via BSON). The access permissions associated with the *publish* data are those specified when the process was defined. In the future, users should be able to both specify output dataset access permissions and define their own processes.

The `connect.remote` and `connect.local.process` capabilities have been designed to be used interchangeably by users. However, capabilities that have been recently implemented for the ARES program were released in the most expedient fashion. As such, source injection is only available to `connect.remote` connections and the capability to handle datasets whose dimensions vary is only available to `connect.local.process` connections. Both of these temporary shortcomings are expected to be resolved in the next release of the client API, which is scheduled for January, 2016.

A `connect.remote` API call that includes a JSON source inject parameter file invokes the source injection capability. Because source injection is dependent upon problem geometry, the wherein the data service

## 2.3 Back end Infrastructure

The BDC backend consists of the following components a) Data Ingest b) Data Management and Extraction c) Workflow Management and d) Web services that provides access to the data. The front-end is implemented in Javascript and the backend is implemented in C++ as Apache modules[16]. The clients and server side communicate via a RESTful API[17].

A C++ library was developed for reading and writing the special HDF5 format used by the BDC data service. The BdcHdf5Accessor library is used in nearly all aspects of the backend infrastructure, from data ingest to data extraction – any time a BDC HDF5 file is read from or written to. An HDF5 file that can be used by the BDC data service requires a very specific structure described by an XML file. The XML file is required to comply with a strict schema, and the BdcHdf5Accessor library validates any XML it uses against the schema.

### 2.3.1 Data Ingest

The data ingest component manages the data coming into the system from various sources. There are two primary ways that a BDC database receives data a) offline registration and b) client registration. Offline registration is the typical way in which previously collected sensor data is registered into the database. Client registration occurs when a client API connects to the BDC via a connect\_remote command and publishes chunked dataproducts to the server, which are then automatically written to an HDF5 file and registered within the BDC.

Both registration methods create indexes of the HDF5 data using FastBit, an efficient compressed bitmap index technology that enables the rapid retrieval of data from the HDF5 file[18]. FastQuery is an extension of FastBit developed at LBNL that allows arbitrary range conditions on the data values contained that span multiple datasets using the FastBit bitmap indices to accelerate the query[19].

The first step in registering new data is to define the hierarchical structure of the data in an XML file. At the bottom of this structure is the dataset, which can be thought of as an N-dimensional array of one kind of data<sup>1</sup>. One or more datasets must be accompanied by an array of Unix epoch timestamps, which have the same length as its sibling datasets (i.e., the sibling datasets are all time-synchronized). If the datasets are also spatially synchronized, optional spatial datasets can be specified giving the x, y, and/or z positions of each data point (e.g., longitude, latitude, and altitude). A collection of sibling datasets and their timestamps is called a datastream. One or more datastreams belongs to a sensor. An example of one sensor with two datastreams, each with multiple datasets, is shown below:

- INS [sensor]
  - NavSolution [datastream]
    - \* Pitch [dataset]
    - \* Roll [dataset]
    - \* Timestamps [timestamp dataset]
    - \* TrueHeading [dataset]
    - \* Path [spatial sync group]
      - Altitude [z dataset]
      - Latitude [y dataset]
      - Longitude [x dataset]
  - SystemStatus [datastream]
    - \* PositionFigureOfMerit [dataset]

<sup>1</sup>Additional handling of cases where N is not fixed has also been implemented

- \* Timestamps [timestamp dataset]
- \* VelocityFigureOfMerit [dataset]

This hierarchical structure is summarized in an XML file for use with the BdcHdf5Accessor library. There are other attributes that must be defined in the XML for each sensor, datastream, and dataset. To make the process easier, a Python code has been developed where the user can easily build the desired structure and output a compatible XML file.




Besides the structure, there are other requirements on HDF5 data that are ingested into the BDC data service. Any ‘timestamp’ datasets must contain monotonically increasing values, otherwise there can be errors or repeated data when it is extracted. For the same reason, two HDF5 files with data from the same sensor cannot have overlapping timestamps. In practice, the process of preparing data for ingest is not trivial and requires some analysis to be performed to enforce these constraints.

### 2.3.2 Data Management and Extraction

The data management backend in BDC consists of various components including the database and stored procedures that are used to manage the data. The SQL database (MariaDB in latest release of BDC) stores the metadata i.e., the information on the runs, sensor information and where the data can be found. The raw data is stored as HDF5 on the filesystem.

The database and the Fastbit indexes help locate the relevant portions of the data in the HDF5 file. HDF5 stores multiple data products/data sets from a sensor system (or logical group of sensors) in a single file. FastBit provides indexing of HDF5 files and FastQuery supports identifying and reading data subsets based on range queries, e.g., time ranges or other filters that are automatically mapped to time ranges. Data is most conveniently accessed in terms of runs (e.g., a single RadMAP test). However, a contiguous measurement can continue through consecutive logical tests, resulting in a datafile that is associated with multiple runs, or a recording can be interrupted during a single run resulting in one run associated with multiple files. So in general there is a many-to-many relationship between runs and files. Furthermore, this correspondence can be different for different sensor systems and their associated files. FastBit and FastQuery are configured to handle these non-specific relationships and to quickly find the portion(s) of file(s) corresponding to requested indexes.

The data extraction component comprises two primary phases that support the web interface and a separate model that enables API connections. During the first phase (query), the data service determines the contiguous time windows during which the user has access to the requested data and then in the second phase the data is extracted from the files. The query phase manages the different ways users can request data. BDC supports Named Queries and Conditional Queries. The query phase determines the time window to be queried. The next phase reads data rows from the HDF5 files corresponding to the time window, which may span multiple HDF5 files. In this phase, data access decisions such as addition of linked datasets like timestamps or spatial synchronization datasets are handled. The extracted data is then bundled together for access by the user. The file is then made available to the user for download through the web interface.

The entire BDC schema is shown in Figure 12. The database software is MariaDB[21], which is an open source MySQL-based database. Arrows pointing from a table (  ) indicate optional (  ) or required links (  ) to another table. For example, the RunSubBlockLink table requires that the Run and SubBlock tables be populated before a link between relating a Run to its data can be created, which is achieved by populating this table. The KEY icon is placed next to the datatype that defines each table entry. Additional required columns within the table are shown as solid blue diamonds. Optional internal column entries are empty blue diamonds. Required cross-table links are indicated with solid red diamonds and option cross-table links as empty diamonds. In the following, subsets of the database are described in further detail.



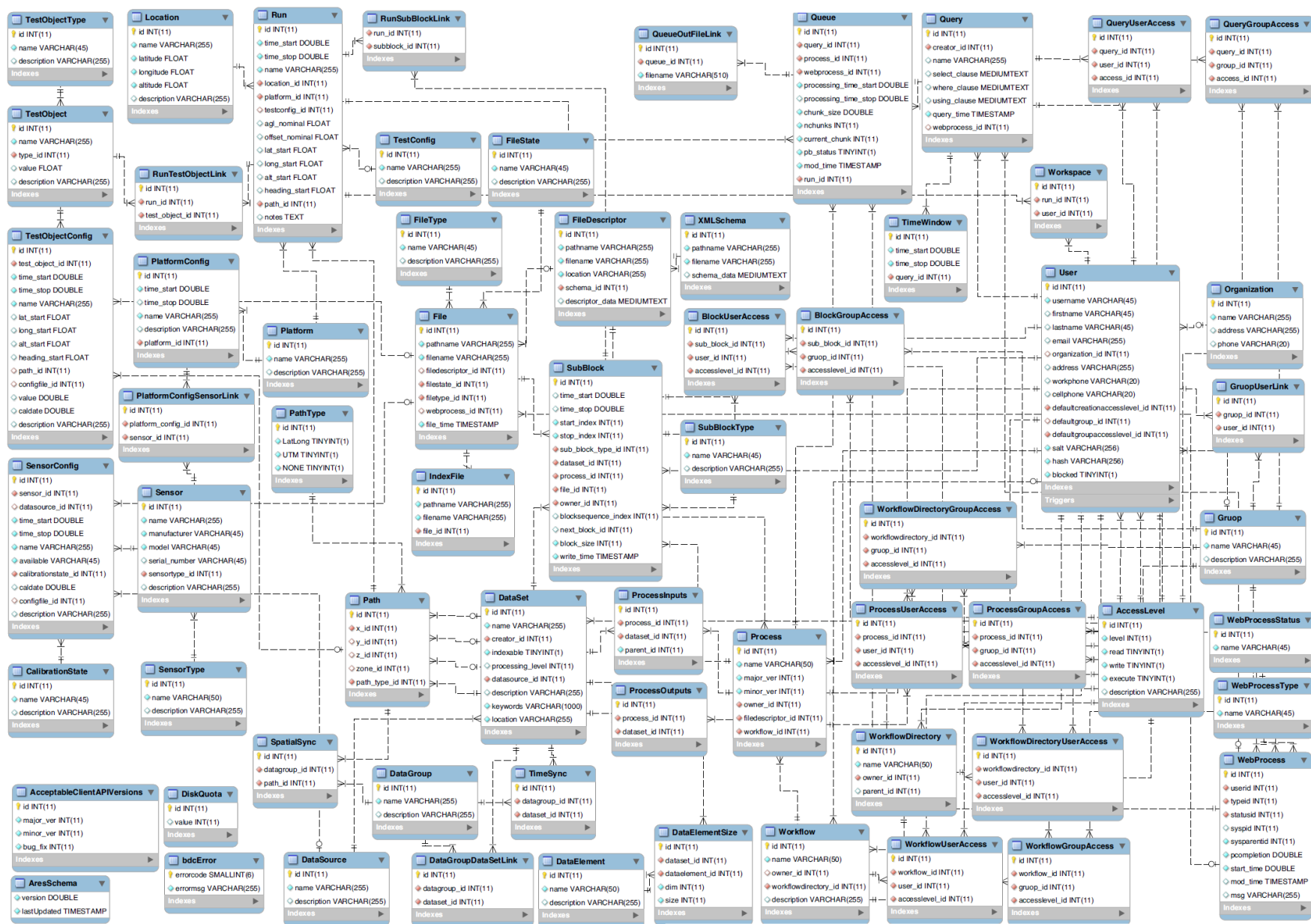
The BDC database has four major functions:

1. Platform and Run configuration management;
2. Sensor data organization and retrieval;
3. Process/workflow management; and
4. User ownership and permissions.

#### **2.3.2.1 Platform and Run Configuration Management**

Management of the configuration of runs and platforms is important to convey to data users which sensors were operating, their calibration status, and the configuration of the data collection environment such as whether a test object (i.e., an intentionally-placed radiological source) is present. Data (by sub-blocks) are associated with a Run and data types and configurations are associated with sensors fielded upon a platform. The database is additionally designed manage configuration files for both sensors and test objects. The below figure shows a portion of the database and has highlighted in red the tables associated with managing runs and run configurations. Runs require a platform, path and location, the latter to facilitate wide-area filtering. Runs can optionally accommodate test objects through a link table, and be associated with a specific configuration. They are most importantly linked to data via the RunSubBlockLink table.

Platforms are associated with runs and are linked by the PlatformConfig table to sensors, which must have a configuration and a definition of the sensor type. The SensorConfig table can link to a DataSource, which provides a sensor-specific link to a data type and therefore the data. Each Sensor must belong to exactly one Platform, which must also be separately registered in the database (the linking between Platforms and Sensors is done using the PlatformConfig and PlatformConfigSensorLink tables.) A Platform can have more than one sensor, and it is meant to represent a system of sensors that have a logical or spatial connection – e.g., disparate sensors that are all on one vehicle. A Run can belong to only one Platform.



**Figure 12.** The BDC database schema.

**Table 1.** DB return

id	name	description
1	DataFile	HDF5 File Containing Measurement Data
2	ModelFile	HDF5 File Containing Modeling Results
3	Downloadable	HDF5 File Used For Downloading Data
4	TestObjectConfig	Arbitrary format File to describe Test object configuration
5	SensorConfig	Arbitrary format File to describe Test object configuration

A specific example defining the test object configuration for a particular run follows.

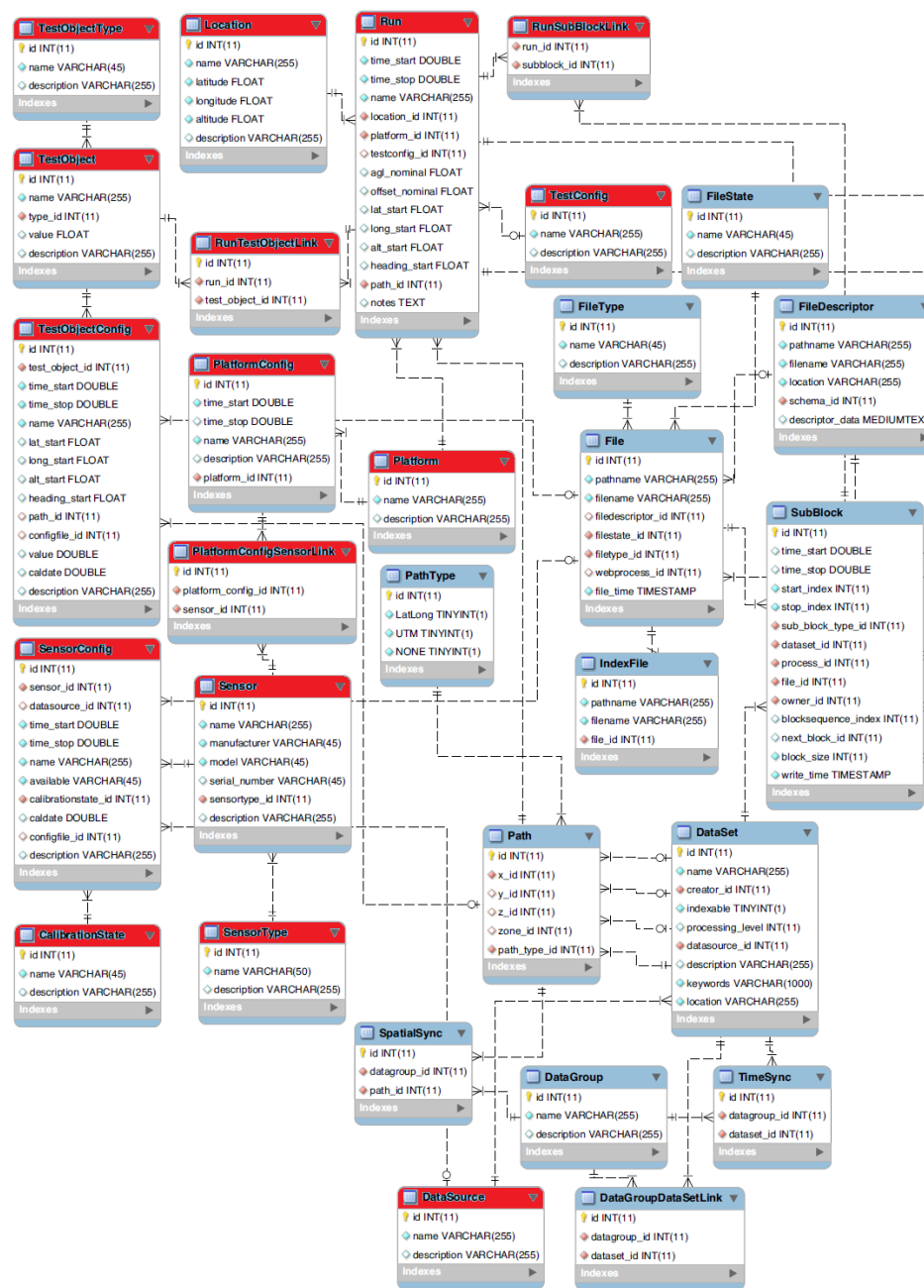
1. Create and register in the database a TestObjectConfig File. The file may be as simple as a word document, or an XML describing the configuration. The FileType table lists all types of Files known to the database. The database query: `SELECT * FROM FileType;` produces the following output. MariaDB [grdc\_db]> `SELECT * FROM FileType;` id4 corresponds to a TestObjectConfig file. Insert a new entry in the File Table, define File.pathname, File.filename and File.filetype\_id=4.
2. Define the test object within the TestObjectType table. Provide a name and optionally a description. E.g., `TestObjectType.id=1; TestObjectType.Name="137Cs button";`
3. Populate the TestObject table with the test object type ID. E.g. `TestObject.id=12; TestObject.type_id=1; TestObject.name="RSL_137-Cs_34059"; TestObject.description="RSL 137Cs Source #34059, 56uCi, calibration date 12/5/2013"`
4. Populate the TestObjectConfig table (note this requires that the TestObject table is first populated and optionally that the TestObjectConfig File is already registered within the database). This is most useful for defining when the TestObject – described by the TestObjectConfig File – is present and where it is positioned, as well as potentially test object motion (via the path table) and/or calibration information.
5. Link the TestObject to a Run by creating an entry in the RunTestObjectLink table.

### 2.3.2.2 Sensor data organization and retrieval

When a BDC-compatible HDF5 file is registered to the data service, the file and its FastBit index file are stored in the File and IndexFile tables, respectively. The XML configuration of the HDF5 file is written to the FileDescriptor table; if the same version of the same XML file is already registered its contents are checked against the stored XML, and registration is aborted if they do not match. The XML schema file is also stored in the XMLSchema table, whose contents are also checked for compatibility.

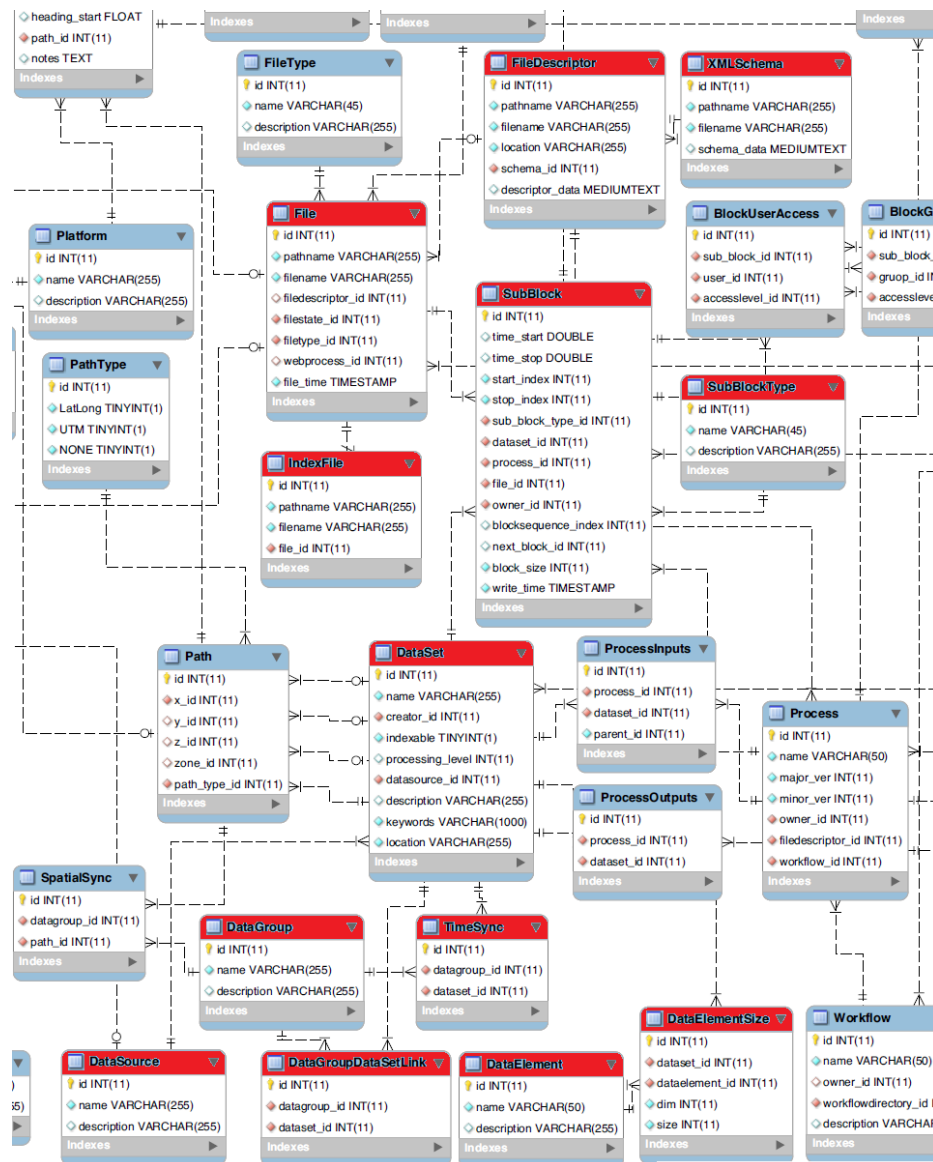
The sensors, datastreams, and datasets in the HDF5 file are also entered into tables, unless entries for them already exist. Each dataset in the file is entered into the DataSet table (unless an entry already exists), and each datastream is entered into the DataGroup table. For each datastream/DataGroup, the required timestamps dataset is linked to the DataGroup via the TimeSync table, and the spatially synchronized data are linked via the SpatialSync table. The sensor is entered into the DataSource table, and linked to the Sensor table if an identical sensor name exists there. The DataSets are linked to the sensor/DataSource that they belong to. The type (e.g., double, float, int64) and number of bytes for each DataSet element can be found in the DataElement and DataElementSize tables.

For each dataset in the HDF5 file, a SubBlock is created in the SubBlock table. The sub-block is the fundamental unit of BDC data – it contains the file the dataset is in, the dataset, the start time, the stop time, and information on how the file was created. If a Run is registered that has a time range



that overlaps the time range of the SubBlock, and if the Runs Platform contains the Sensor linked to the SubBlock (via the DataSource table), then the Run and SubBlock can be linked using the rsblinkcalc utility. Once that link has been made, the data from the SubBlock will be available when querying the Run.

16



**Figure 14.** A portion of the BDC database schema with tables associated with data organization highlighted in red.

Data are retrieved from a BDC database via queries. Each query populates the Query table within the database. Queries are either generated from the web-based interface or by an API connection. Queries are always associated with a user and require a time window, which enables the BDC to take advantage of the FastBit indexing. The query time window may be the extent of the Run, which is the default behavior if no window is specified. Tables associated with data retrieval are highlighted in the figure below. Additionally, the connections from the WebProcess table to the Query and Queue tables are highlighted for clarity. The optional connection from the WebProcess table that is not contained within the figure goes to the File table.

All queries that are generated by users (as opposed to debugging queries) are created by a WebProcess. These can include Drupal web interface queries such as direct queries for file download and local process file queries that produce files for connect\_local\_process API connections. The small incremental queries that are automatically generated by the connect\_remote API calls and connect\_local\_process Drupal queries are managed in the database using the Queue table, which requires that a process be specified via the

The diagram illustrates a comprehensive database schema for a workflow management system. It includes tables for user management (User, Organization, Group, GroupUserLink), workflow execution (Run, Queue, Query, QueryUserAccess, QueryGroupAccess, Workspace, TimeWindow), file and schema management (FileDescriptor, XMLSchema), access control (BlockUserAccess, BlockGroupAccess, AccessLevel, ProcessGroupAccess, ProcessUserAccess, WorkflowDirectoryGroupAccess, WorkflowDirectoryUserAccess, WorkflowGroupAccess, WorkflowUserAccess), and workflow components (SubBlock, SubBlockType, ProcessInputs, ProcessOutputs, Process, Workflow, WorkflowDirectory, WorkflowUserAccess, WorkflowGroupAccess, DataElementSize, DataElement). Relationships are defined using primary and foreign keys, often with cardinality indicators like '1' and 'N'.

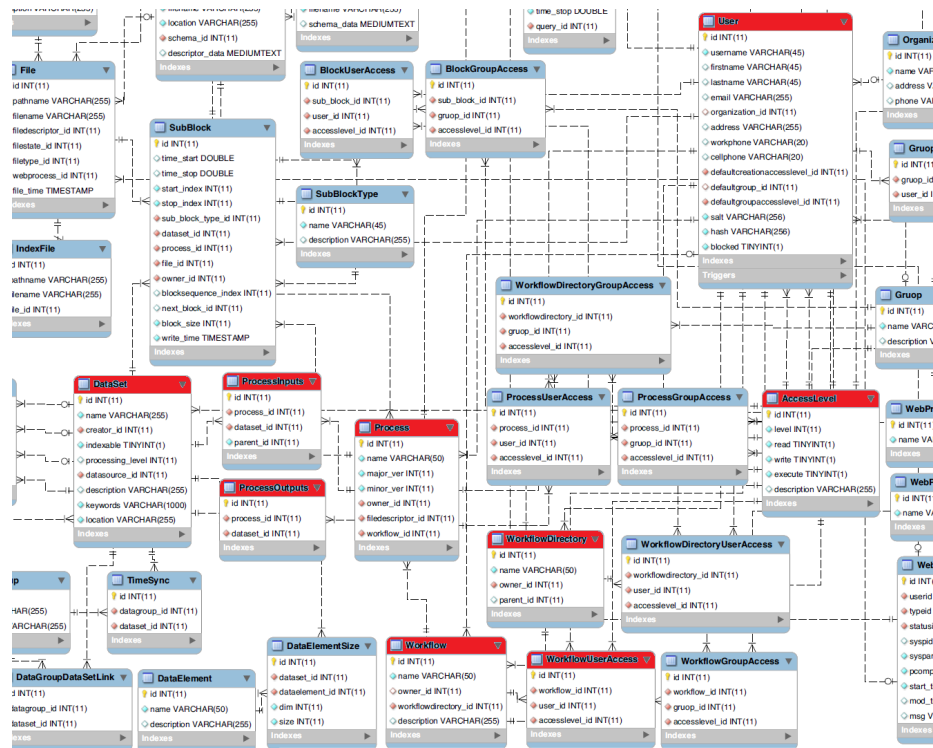
**Figure 15.** A portion of the BDC database schema with tables associated with data retrieval highlighted in red.

A BDC database considers a Process as a list of input and output datasets that are contained within a workflow and owned by a particular user. If for a particular run, all required input DataSets are available to a user and the user has access permissions to the process, the user may execute an algorithm such that the BDC treats algorithm outputs as those specified by the process.

18



workflows. A workflow is simply a container for a series of related processes. For example, a detection algorithm workflow could first involve a spectrum-generating process that takes list-mode gamma-ray data and INU positioning data as inputs and creates spectra whose time-widths are determined by the platform velocity. A second process could take those spectra as inputs and output an anomaly detection score, which then is input to a third process within the workflow that uses platform positioning and the score to localize the anomaly and to either associate it with a stationary point-like source, a dynamically moving source, or a benign environmental condition. Beyond the connection to data management via Process Inputs and Outputs, Processes and Workflows are primarily linked within the database to permissions management capabilities.



**Figure 16.** A portion of the BDC database schema with tables associated with process and workflow management highlighted in red.

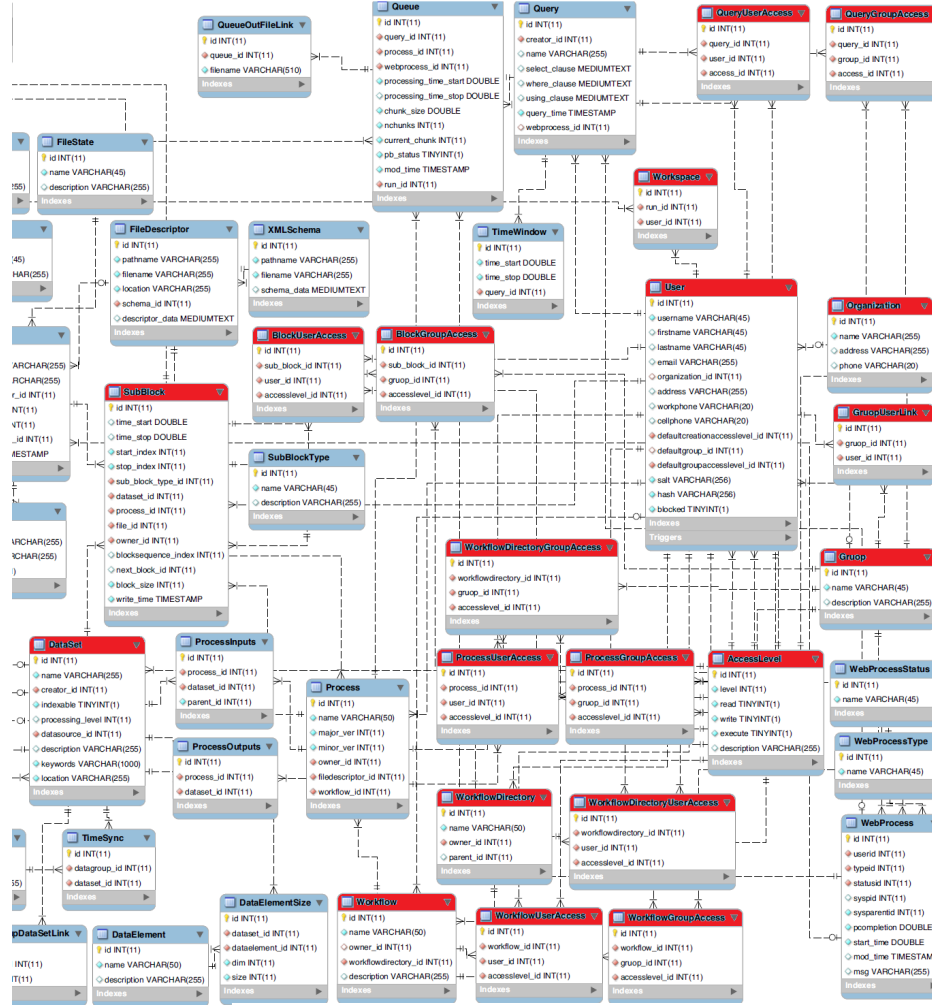
#### 2.3.2.4 User ownership and permissions

Data ownership and access permissions are set at the subblock level. Each subblock has an owner whose user\_id is tabulated in the User table. Access is managed within the BlockUserAccess and BlockGroupAccess tables. The BlockGroupAccess table enables an entire group's permissions to be assigned without explicitly assigning each users' access level because users are assigned to groups via the Group table (the table name is a typo within the database). The AccessLevel table specifies access levels akin to Linux read/write/execute permissions.

Users that do not have permission to read a particular subblock can neither access the data via the Drupal webpage, nor see that the data exists within the query and workspace settings applications on the webpage. Users will additionally be unable to open connect\_remote API connections or generate connect\_local\_process files if the process they attempt to use requires input data for which they do not have permissions to read.

Workflows, workflow directories, processes, and queries are all managed using the same permissions configuration. Users who do not have permission to access a workflow directory will not be able to

connect via API to any processes within the workflow directory, nor will the website show workflows in that directory when accessing the ProcessFile Generator page (i.e., Figure 8). Similarly, if a user has access to a workflow and workflow directory, but not a process, that process will not populate the Processes table on the the ProcessFile Generator page in the website. If a user has read but not execute permission to a process, they would be able to download data and connect\_local\_process files, but could not register output data to the BDC via a connect\_remote API connection.



**Figure 17.** A portion of the BDC database schema with tables associated with permissions management highlighted in red.

### 2.3.3 Workflow Management

The workflow management in BDC is handled through three main components. The process and workflow definitions are stored in the database (see refSS:ProcMgmt). The workflow service (described below) and the corresponding client interfaces accomplish the database management. Workflow execution in BDC is managed through Tigres[20]. Tigres is a workflow library that has Python and C interface. Tigres supports the concept of composing the workflows in the form of templates, standard building blocks (similar to MapReduce). Currently Tigres supports four templates – sequence, parallel, split and merge. Tigres provides support for running analysis workflows on desktops and clusters and provides provenance, monitoring and user-logging capabilities. In BDC, the client library is used to query the database and



**Table 2.** Functions supported within the BDC rest API.

HTTP method	URL
GET	workflow/dataset?id="/Data/Set/Name"
POST	workflow/dataset?id="/Data/Set/Name"
DELETE	workflow/dataset?id="/Data/Set/Name"
GET	workflow/admin/group
POST	workflow/admin/group
PUT	workflow/admin/group
DELETE	workflow/admin/group?id=GROUP_NAME
GET	workflow/directory?id=DIRECTORY
POST	workflow/admin/directory
DELETE	workflow/directory?id=DIRECTORY
GET	workflow/process/permissions?id=PROCESS&type=user
POST	workflow/process/permissions?id=PROCESS&type=user
DELETE	workflow/process/permissions?id=PROCESS&group_id=LBL
GET	workflow/process?id=PROCESS
POST	workflow/process
DELETE	workflow/process?id=PROCESS
GET	workflow?id=WORKFLOW
POST	workflow
DELETE	workflow?id=WORKFLOW

generate a Python Tigres program that defines the workflow process and dependencies. The user is able to tweak the workflow to include process implementations (i.e., executables and functions). Tigres programs can be submitted as jobs to batch queue systems. These are currently developed and tested for both the GRDC environment and the ARES environments.

### 2.3.4 Web Services

The Apache web services provide the gateway to the backend and the data for end-user use. The web service is accessible through a REST interface. REST architecture involves exchanging information from the server to client through an XML file that describes and includes the desired content. BDC provides two web services: a) Data Service and b) Workflow Service. The Data Service can be used to query and get data as well as to manage the workspace (insert, delete) and browse dataproducts and workflows either through the client API or a web browser. The functions supported by the data service are shown in Table 3. The workflow service allows the users and special users (administrators) to list/show workflows, grant permissions, and to create and update workflows. The workflow database support in the workflow service is provided using ODB[22]. ODB provides a cross-platform and cross-database object-relational mapping (ORM) system for C++. It allows us to persist and read C++ objects to a relational database.

## 2.4 Interaction with frontend

Data extraction is driven by user interaction in the browser. Users are allowed to manage a list of runs they are actively working on in the workspace manager. This enhances usability by allowing users to manage a working set but also facilitates BDC performance optimizations by restricting the amount of information transmitted to the front end. To select runs of interest, the back end transmits a table of runs

**Table 3.** Workflow management functions supported within the BDC rest API.

HTTP method	URL	Notes/comments
POST	bdc/query?mode=1	Local Download
POST	bdc/query?mode=2	Local Process File Download
POST	bdc/query?mode=3	Realtime Mode
GET	bdc/query?wpid=WPID	Non blocking, return immediately
GET	bdc/file?wpid=WPID	
GET	bdc/status	
GET	bdc/workspace/runtable	
POST	bdc/workspace/insert	
POST	bdc/workspace/delete	
GET	bdc/browse/histogram	
GET	bdc/browse/dataproduct	
GET	bdc/browse/workflow	
GET	bdc/browse/dataproduct?run=RUN1;RUN2;..	By default return the Lat, Long, Time correlated values for the selected data products formatted in a JSON string
GET	bdc/browse/dp-sample	
GET	bdc/queue/nextChunk?wpid=WPID	
POST	bdc/queue/publish?wpid=WPID	
GET	bdc/auth/token	
POST	bdc/auth/token	

associated with information about start time, duration, platform, system, streams to the user. Users can filter, select, add, remove runs from the workspace that triggers the appropriate actions in the backend

As discussed earlier, the data browser provides the interface to get an overview of one or several runs by showing the flight path on a map, displaying histograms of data products and supporting the formulation of range queries based on path and data product values (e.g., time ranges, altitude range). The frontend and backend interact to obtain a list of all data products for a run along with detailed information (data type, size, dimensionality) from the back end. Once the user has selected one or more runs, the path information for the run(s) (Latitude, Longitude, and Altitude, Timestamps) from the back end are obtained. The information for the position and time syncing and run start and end times are also obtained from the backend database. The FastQuery/FastBit indexes are then used to read the portions of the HDF5 files, corresponding to the runs of interest. The files are then encoded in JSON and sent to the web interface. The information is then decoded on the frontend to display information on a map view and the users can add histogram for any data product associated with the runs. If a dataproduct is not available on the frontend by default, a RESTful query is executed to obtain the data from the backend. The backend queries the BDC database to find start/stop times of runs, filenames for datasproducts, names of the corresponding timestamp datasets (for synchronizing to position data and identifying subsets within run time range). FastBit/FastQuery are used to read appropriate data subsets and to provide them to the frontend. The frontend uses timestamps to synchronize data with positions and possibly other loaded datasets. Additional data products can be used to color flight path (to show value variation during run) and to display histograms. Selecting regions in the map and ranges in the histogram modifies a range query string that corresponds to the currently selected data portion. The user can submit a query with that string to the backend, which then creates an HDF5 data set for download

### 3 GRDC Status

#### 3.1 Capabilities

By storing and providing data in HDF5 formats, BDC is facilitating management and analysis of advanced detector types that produce more complex data than can be managed in existing file format standards. HDF5 is also optimized for parallel and scalable access that facilitates scalable data access and analysis in data models where data volumes continue to grow. BDC uses advanced metadata to describe sensors, the data they generate, and analysis dataproducts and maintains a hierarchical data model that describes the workflows, processes, runs and dataproducts.

The Gamma Ray Data Cloud code base has been effectively managing and disseminating RadMAP and AMS data. The BDC software has been developed such that analyses and algorithms can be developed and tested in a local setting before being applied to previously withheld datasets in a fashion that simulates real-time data collection such that algorithms can be tested on real data. ARES-driven developments to enable source injection and data visualization could readily be generalized to have wider impact on GRDC.

#### 3.2 Data Available on GRDC

At present, RadMAP data registered within the GRDC include all HPGe and all NaI data from collections between November 12, 2011 to October 29, 2013; Liquid Scintillator neutron data are available starting August 30, 2012, three positioning datagroups: the Magellan GPS/IMU, the Novatel GPS/IMU, and the vehicle speed diagnostics system; and data produced by the two weather sensors: a NewMountain100 station operated from briefly between April and May of 2012 and a VantageVue station, which was installed in December of 2012. Three of the four vehicle positioning datagroups are available for all data campaigns that have been made available on GRDC. The Magellan GPS has been recently removed from the RadMAP system, as its functionality has been replaced by the Novatel GPS/IMU, which was installed prior to the January 28, 2013 measurements.

The HPGe datagroup includes calibrated list-mode energy deposition events, the system times of the events, and the detector ID of the event (there have been up to 28 separated HPGe detectors fielded within RadMAP at once). Much of the HPGe data had initially displayed spurious low-energy noise events that exhibited a broad range of amplitudes, yet peaked near an amplitude equivalent to 150 keV. These were found to be due to the digitizer system malfunctioning. The noise events were found to be correlated in time. Additionally, digitizer channels that were not connected also showed the noise phenomenon. As such, the disconnected channels (which should contain no events) were used to time-filter the noise events from the connected channels. This procedure is explicitly summarized in Ref. [23]. Typical list mode event rates are 1000 events/second.

The NaI datagroup includes calibrated list-mode energy deposition events, the system times of the events, and the row and column of the detector that underwent the interaction within the 10x10 NaI array. Muon events that interacted with multiple NaI detectors were used to demonstrate synchronization across the NaI array. This is described in further detail in Ref. [23]. Typical list mode event rates are 12000 events/second.

The liquid scintillator detector array was provided as part of the RadMAP system by Sandia National Laboratory (Livermore). It consists of 16 separate detectors, which are sensitive to both neutrons and gamma rays, but the time-dependent characteristics of the event pulse can be used to determine whether the event was due to neutron or gamma-ray interaction. As such, the digitized waveforms for each event are provided along with the timestamp of the event, the detector id, and the integrated pulse amplitude. Event rates are approximately 2200 events/second.

The Magellan GPS unit was the legacy positioning capability from the MISTI system. When operating

without GPS signal interruptions, it produces a 6 Hz output that describes RadMAP position and stance. The INS was not used to inform the GPS solution, which was found to produce positioning data of insufficient accuracy for many analyses envisioned for RadMAP data particularly in urban settings where GPS signals can be unreliable. Despite this, the system continued to be operated until October, 2015 because it also provided the system clock and trigger signals for many contextual sensors. Additionally, the data could be used in conjunction with the other positioning datasets to inform GPS attenuation and reflection models, should a researcher with interest in developing software to improve GPS performance desire such a dataset.

The Novatel INS contains differential GPS coupled with an inertial motion unit (IMU) in order to produce better positioning outputs, particularly in GPS-starved environments. The datagroup contains positioning, stance, and positioning error datasets at 10Hz. Comparing with Lidar-reconstructions, the positioning data have been found to be accurate to 10-100 cm.

The Vehicle Diagnostics datastream contains the vehicle speed derived from the rate of tire rotations. The datastream is provided at 3.33Hz. The NewMountain weather station produced dew point, heat index, humidity, pressure, temperature, wind speed and direction datastreams. The VantageVue station additionally included datastreams derived from a rain gauge. Both weather systems reported at a rate of once per minute.

Additional RadMAP datastreams exist and are in various stages of maturity with respect to dissemination via GRDC. Lidar data are synchronized and parsed, but no BDC XML schema has been defined. Additional dataproducts resulting from segmentation analyses have been registered but their registration within BDC would be more difficult to optimally implement owing to their potential value as metadata that should be spatially synchronized but not associated with any particular collection time. These same BDC registration challenges are present for dataproducts that would result from a data-fusing analysis that attempts to ‘invert’ measured gamma-ray spectra and attribute them to visible surfaces. Data derived from the hyperspectral (HSI) sensors have been parsed, but synchronization was not achieved until the most recent timing circuitry was implemented. HSI calibration is also incomplete due to the unique complexities associated with land-based HSI measurements in urban environments and the resultant variations in ambient illumination. Data parsing routines that produce still-frames of the high-definition video data (Ladybug and CCD cameras) have been developed. Similar BDC XML schemas have been developed such that registering video still frames to BDC would be trivial, however some additional frontend development would be valuable such that the data could be visualized on the GRDC website.

Some data generated using a helicopter-mounted Remote Sensing Incorporated (RSI) NaI-based gamma-ray detection system has also been made available on GRDC. These datasets comprise measurements collected in the San Francisco Bay Area in 2012 and all datastreams typically generated by the AMS RSI system have been made available including 1-s gamma-spectra, livetimes, radar altimeter, GPS positioning, and system health metrics.

## 4 Outlook

BDC provides advanced capabilities to access and process nuclear radiation data through an open-source and open-access framework. In summary, BDC has the following unique capabilities:

- Seamless access to data from RadMAP, AMS, ARES, HPGe, and other fielded sensors. Some datasets are uniquely available through BDC instances.
- Data is stored in HDF5, a portable data format that is also optimized for parallel scalable access that is critical as data volumes are growing.

- Advanced metadata about the sensors, the data they generate, and analysis dataproducts is stored and used for accessing the data.
- Hierarchical data model that describes the workflows, processes, runs and dataproducts.
- Support for complex workflow definitions and analyses capabilities.
- Support to run scalable analyses on clusters and high-performance computing systems.
- BDC is uniquely suited for side-by-side algorithm comparisons.

BDC provides a strong foundation for the community to access and share data. Additional capabilities and efforts are needed to sustain the development and engage the community widely. A multi-agency measurement campaign at a controlled and well-characterized facility would be an ideal way to demonstrate to the broader community the value of BDC capabilities.

Big data technologies and capabilities are rapidly evolving. This provides the BDC team some challenges and opportunities. Since BDC development started in 2011, a number of new technologies have evolved that have advanced real-time processing and data analyses capabilities. Many of these are very relevant to the needs of radiation detection, radiation detection algorithms, data fusing algorithms, and persistent monitoring applications. The BDC team has wide range of expertise with big data and cloud technologies that provides an unique opportunity to leverage these technologies and further improve the BDC software stack. Additionally, as new technologies become available that combine radiation sensors with contextual data, there is a need to expand to support additional data sources and formats. We identify these and other opportunities for BDC development in the list below.

- Error checking and UI improvements such that users may generate their own processes and manage their workflows. This is currently a capability that is only provided to BDC developers.
- Speed – chunked API data accession is not efficient. Rather than select an entire dataset from a single database query (and the associated file I/O) and then perform chunking by leveraging (the copious available) random access memory (RAM), each chunk is produced through a separate database query and I/O instance. Improving speed is actively being pursued in support of the ARES ATD.
- Streamlining source injection and generalizing the capability – SI was recently developed for ARES and has not been generalized for other platforms and datasets.
- Both managing and enabling visualization of data types that are not readily suited for the paradigm of organizing data by Runs and indexing by time such as purely modeled data or derived data products representing stationary non-ephemeral objects or conditions (i.e., lidar-derived surfaces and their properties).
- Additional API connectivity, particularly outputting BDC data to Monte Carlo codes and accepting Monte Carlo outputs as inputs.
- Enabling a user to specify the format(s) in which queried data products will be delivered (such as N42[24] and CSV). List-mode data (for which BDC was developed) does not have a widely-accepted standard. However data output according to a user-specified chunking time could be provided.
- API connectivity – develop data ingress pipelines that can pull data from other cloud-based platforms such as RadResponder[25].
- Improved database flexibility by transitioning to NOSQL or another modern data framework.

- Streaming data and real-time processing of data.
- More responsive front-end technologies.
- Use of deployment technologies like Docker for managing BDC services.
- Better custom analyses support.
- Handling of optional inputs to processes.

There are other, less technical issues/needs related to GRDC that are listed below.

- Develop a broad user community – both from the perspective of data users (such as algorithm developers) and data producers who benefit from BDC data management capabilities.
- User management issues how long is your data yours?
- The business case for GRDC to become self-supporting such that the initial funders can continue to receive benefits without providing continued maintenance and development support.
- Classified data – nothing prohibits an instance of BDC being deployed on a classified network, however, LBNL is prohibited from handling classified data, therefore a mature and robust version of BDC that could be managed by a trained user would be ideal.

The BDC code base has been deployed to support the Characterization in the DNDO ARES ATD and to disseminate radiation, positioning and stance and weather data via `grdc.nersc.gov`. More recently, NA-42 has begun supporting initial scoping of applications and improvement needs such that BDC could support AMS reachback and CM data management needs. BDC is already a truly unique capability within the nuclear detection and non-proliferation application space, yet its potential reaches far beyond that which has been demonstrated through the initial feasibility study and the very ARES-specific development.

## References

- [1] K.-P. Ziock, K. E. Nelson, “Maximum detector sizes required for orphan source detection,” *Nuclear Instruments and Methods in Physics Research Section A* volume 579, issue 1, (2007), Pp. 357-362.
- [2] D.M. Pfund, K. D. Jarman, B.D. Milbrath, S.D. Kiff, and D. E. Sidor, “Low Count Anomaly Detection at Large Standoff Distances,” *IEEE Transactions on Nuclear Science*, volume 57 issue 1, February 2010.
- [3] Timothy J. Aucott, Mark S. Bandstra, Victor Negut, Joseph C. Curtis, Daniel H. Chivers, and Kai Vetter, “Effects of Background on Gamma-Ray Detection for Mobile Spectroscopy and Imaging Systems,” *IEEE Transactions on Nuclear Science*, volume 61 issue 2, April 2014.
- [4] Timothy J. Aucott, Mark S. Bandstra, Victor Negut, Joseph C. Curtis, Ross E. Meyer, Daniel H. Chivers, and Kai Vetter, “Impact of detector efficiency and energy resolution on gamma-ray background rejection in mobile spectroscopy and imaging systems,” in *Nuclear Instruments and Methods in Physics Research A*, 789 (2015) pp. 128-133.
- [5] Timothy J. Aucott, Mark S. Bandstra, Victor Negut, Daniel H. Chivers, Reynold J. Cooper, and Kai Vetter, “Routine Surveys for Gamma-Ray Background Characterization,” *IEEE Transactions on Nuclear Science*, volume 60 issue 2, April 2013.

- [6] T.H. Joshi, R.J. Cooper, J. Curtis, M. Bandstra, B.R. Cosofret, K. Shokhirev, D. Konno, “The impact of energy resolution, detector array size, and detection algorithm on threat detection sensitivity of mobile systems,” *Submitted to IEEE Transactions on Nuclear Science, October, 2015.*
- [7] Drupal Association <https://www.drupal.org/>.
- [8] SimpleGeo and Stamen, <http://polymaps.org>.
- [9] Microsoft Inc., <https://www.bing.com/maps/>.
- [10] Square Inc., <http://square.github.io/crossfilter/>.
- [11] Mike Bostock, <http://d3js.org>, <https://github.com/mbostock/d3>.
- [12] jQuery Foundation, <https://jquery.com/>.
- [13] The HDF Group and the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign, <https://www.hdfgroup.org>.
- [14] <http://www.json.org/>
- [15] <http://bsonspec.org/>
- [16] <http://httpd.apache.org/>
- [17] <http://www.restapitutorial.com/lessons/whatisrest.htm>
- [18] K. Wu et. al, “FastBit: interactively searching massive data,” *Journal of Physics: Conference Series*, 180 (2009) 012053.
- [19] (<http://www-vis.lbl.gov/Events/SC05/HDF5FastQuery/>)
- [20] L. Ramakrishnan, S. Poon, G.Z. Pastorello, D. Gunter, V. Hendrix, D. Agarwal, “Scientist-Centered Design for eScience:A Tigres Case Study”, IEEE eScience, 2014. <http://tigres.lbl.gov>
- [21] <https://mariadb.org/>
- [22] <http://www.codesynthesis.com/products/odb/>
- [23] M.S. Bandstra et. al, “The Radiological Multisensor Analysis Platform: a Tool for Exploring the Impact of Contextual Information on Mobile Radiation Sensors,” *to be submitted to NIM A January, 2016.*
- [24] “American National Standard Data Format for Radiation Detectors Used for Homeland Security”, “ANSI N42.42-2012 (Revision of ANSI N42.42-2006)” April 15 2013 *IEEE Standards*.
- [25] ChainbridgeTech Inc., <https://www.radresponder.net/>.

## A Appendix A - Glossary

- Client: software that interfaces with BDC via external connections by including BDC libraries and invoking a connect command.
- Platform: the system on which one or more sensors are fielded to collect data.

- Run: a continuous time range for which sensors were deployed on a Platform collecting data (gaps and dropouts can occur).
- Datastream: a contiguous series of data either collected with a sensor or resulting from an algorithm.
- Dataproduct: a Datastream that results from an analysis Process.
- Synchronization: implies sensor data and resulting Dataproducts that are explicitly correlated in time and result from the same Run.
- Data Element: a unit of data, may a single datum or may be multi-dimensioned, both the energy of a detected gamma ray and a frame of video data constitute a data element.
- Datagroup: a set of Synchronized Datastreams that have the same number of elements because they are collected on the same Platform with the same periodicity.
- Query: the software equivalent of requesting Data from the BDC. Queries may span across Runs and platforms, but only in the case where identical Datastreams have been requested would the Query produce datasets that span across different platforms.
- Chunk: a discrete set of data that is delivered by the BDC. The chunk width defines the time interval duration over which all requested synchronized data will be delivered.
- Process: An abstract representation of the database definitions associated with retrieving data or generating new datastreams within the database. Processes are uniquely identified by their name, version information and Workflow. Each process also maintains information about input and output DataSets.
- Workflow: The abstract representation of a Process execution context. A workflow can be used to represent a series related processes. Workflows are uniquely identified by a name and a workflow directory.
- Process Registration: defining all required inputs and the format of each output and creating a process in the BDC database.
- Data Registration: providing BDC with expected outputs from a Registered Process.
- Process File: an HDF5 file that enables BDC process connectivity to be simulated locally.
- Source Injection: the insertion of additional non-benign signatures into datastreams that are nominally considered benign.
- Test Object: an object associated with creating non-benign signatures in simulation or by measurement.

## B Appendix B - BDC User Manual





---

# Berkeley Data Service Programmatic API (Version 1.2.2)

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

---

ARES Characterization Team

# Table of Contents

---

Berkeley Data Service Programmatic API (Version 1.2.2) .....	1
Introduction .....	1
API Operational Modes – Two Use Cases .....	2
Local File Mode .....	2
Remote Server Mode .....	3
Function Descriptions .....	4
Installation Instructions.....	12
POSIX Operating systems (Linux and Mac): .....	12
Windows Server: .....	12
Wrappers for the Programmatic API .....	13
Python:.....	13
Java:.....	13
Matlab:.....	13
Quick Reference - API Functions .....	15
Glossary .....	<b>Error! Bookmark not defined.</b>

# Berkeley Data Service Programmatic API (Version 1.2.2)

## Introduction

---

This document describes the Application Programming Interface (API) developed to facilitate interaction with the Berkeley Data Service, also referred to as a Data Cloud. The programmatic interface provides a pseudo-real-time view of the data stored by the Data Service, enabling straightforward development and testing of analysis routines. For the ARES program, Topic 2 vendors will use this API to retrieve data for analysis and must use it to register and record the algorithm outputs within the Data Service. In the ARES Characterization Phase, the Characterization Team will pipe test data to these Topic 2 algorithms for the purpose of characterizing the ARES system.

To address the needs of both the Topic 2 vendor analysts and integrators, the API is available as a C-Language library, with wrapper interfaces for Python, MatLab and Java environments. The API provides pseudo-real-time access to data by two modes. In the Local File Mode, the API provides data to the algorithm incrementally by accessing a local file, which has been downloaded from the `ares.lbl.gov` webserver. In the Remote Server Mode, the requested data is incrementally accessed and transmitted by the Data Service and made accessible through the API on client machine. Only the Remote Server Mode can be used to record and report results. Irrespective of the data source, however, the core functionality and workflow is constant in both operational modes.

The text in the remainder of this document is formatted to highlight a number of features. Unique terms and concepts are presented in `Courier` (ex. `Data Service`) and documented in the Glossary. API functions and subroutines are formatted in **Bold**. Emphasis is denoted using *Italics*.

## API Operational Modes – Two Use Cases

---

### Local File Mode

Users are currently able to view a catalogue of data available in the `Data Service` via a web-interface on the `ares.lbl.gov` server. This web-interface facilitates user queries of the service and provides a means for bulk data downloads in the form of HDF5 files. The Local File Mode of the API discussed in this document enables a means of accessing these HDF5 files to test pseudo-real time algorithm functionality. Results generated from data sourced via these HDF5 files cannot be recorded within the `Data Service`. This functionality is only available to Remote Server Mode connections.

Starting with version 1.1 of the API, users can record results to a local output file if the `Data Service` has specially prepared the downloaded input file. The files are accessible to users from a Web-based User Interface. These specially prepared files contain metadata generally describing what the user's program will do. These metadata describe the `DataSets` the program intends to consume as input, the intended outputs, as well as the context in which these `DataSets` are to be referenced and saved. In Local File Mode, users transparently reference these metadata with a string-based tag, referencing a pre-registered `Process`. Tags are formatted to represent the `Process` name, major version number, and minor version number. In the tag, the name is delimited from the version number by the symbols '<' and '>', while the version numbers are delimited by a period (`RSI_GROSS_COUNTER<0.1>`). Documentation on how `Processes` are registered will be communicated at a later date.

*The additional functionality provided by version 1.1 is intended to augment, rather than replace that originally distributed in version 1.0. The updated library supports both implementations.*

In the Local File Mode accessible from version 1.0 of the API, users initialize and access the data through a series of steps:

1. Initializing a session by calling the **`connect_local()`** subroutine.
2. Sequentially loading `Chunks` into memory by calling the **`getNextChunk()`** subroutine.
3. Accessing the data from each `Chunk` by calling **`getNumElem()`** or **`getDataElement()`** as appropriate.
4. Enabling access to the next `Chunk`, by calling the **`publish()`** subroutine.

5. Cleaning up the session when complete by calling the **disconnect()** subroutine.

In the Local File Mode accessible from version 1.1 of the API, users initialize, access and write data through a *very similar* series of steps:

1. Initializing a session by calling the **connect\_local\_process()** subroutine.
2. Sequentially load data `Chunks` into memory using the same calls as in version 1.0.
3. Access data from each `Chunk` using the same calls as in version 1.0.
4. Optionally stage data for writing using the appropriate version of the **stage\*()**. Here the asterisk symbol represents a wildcard to be replaced by relevant terms from the detailed functional description below.
5. Write data to the output file and enable access to the next `Chunk`, by calling the **publish()** subroutine.
6. Cleaning up the session when complete using the same call as in version 1.0.

Each of these functions is documented in the section entitled Function Descriptions, below. Following the workflow described above provides data in a pseudo-real-time format similar to that available in the Remote Server Mode.

### Remote Server Mode

The implementation of this mode is still in progress and the document will be updated when work is complete. With this caveat in mind, a preview for the Remote Server mode is described below. Associated function descriptions whose implementation is subject to change are noted as such. However, this uncertainty should not impact users' ability to access the data, architect, or tune algorithms. Please let the Characterization Team know if this is in fact not the case.

Similar to the use case discussed for the local file mode, users access the data following a multi-step workflow. In the Remote Mode, two additional steps are present.

1. Initializing a session by calling the **connect\_remote()** subroutine.
2. Sequentially loading `Chunks` into memory by calling the **getNextChunk()** subroutine.

3. Accessing the data from each `Chunk` by calling `getNumElem()` or `getDataElement()` as appropriate.
4. Optionally staging algorithm results to be transmitted to the `Data Service` using the appropriate `stage*()` subroutine. Here the asterisk symbol represents a wildcard to be replaced by relevant terms from the detailed Functional description below.
5. Acknowledge processing of the present `Chunk` and submit results to the server using the `publish()` subroutine. Receipt will be time-stamped by the *end* of the `Chunk`.
6. Cleaning up the session when complete by calling the `disconnect()` subroutine.

## Function Descriptions

---

The following list describes the functions implemented within the API. The syntax used in this section is in the C-Language, but the exact implementation will ultimately depend on the chosen programming environment. Please see the example programs or scripts provided for each environment.

■ **int connect\_local(char \*\*handle, char \*select\_clause, char \*filename, double chunk\_size)**

The function initiates a local file mode session and returns the number of `Chunks` the user may expect to analyze in pseudo-real-time. To determine this number, the specified file is examined and references to the data are stored for each `Chunk`. The time required to determine these references will depend sensitively on the amount of data in the file.

The first argument “handle” is a pointer to memory internal to the API and may not be altered by the user.

The second argument “select\_clause” should point to a comma-separated string of `DataSet` names. See the `DataBrowser` page on [ares.lbl.gov](http://ares.lbl.gov) used to download the HDF5 File or the “dbDataSetName” attribute in the HDF5 file for allowed values.

The third argument “filename” should point to a string containing the full system path of the HDF5 file being sourced for data.

The fourth argument “chunk\_size” sets the time range, in seconds, for each `Chunk`.

- **int connect\_local\_process(char \*\*handle, char \*process\_tag, char \*input\_filename, char \*output\_filename, double chunk\_size)**

The function initiates a local file mode session and returns the number of `Chunks` the user may expect to analyze in pseudo-real-time. To determine this number, the specified file is examined and references to the data are stored for each `Chunk`. The time required to determine these references will depend sensitively on the amount of data in the file.

The first argument “handle” is a pointer to memory internal to the API and may not be altered by the user.

The second argument “process\_tag” should point to a string containing the `Process` name, major version number, and minor version number (`RSI_GROSS_COUNTER<0.1>`). The process information will determine which `DataSets` will be available for analysis.

The third argument “input\_filename” should point to a string containing the full system path of the HDF5 file being sourced for data. This file must contain metadata on the process being requested, or an error will be displayed and the code will exit.

The fourth argument “output\_filename” should point to a string containing the full system path of the HDF5 file into which the output will be written. If the file exists on the system, it will be overwritten.

The fifth argument “chunk\_size” sets the time range, in seconds, for each `Chunk`.

- **int connect\_remote(char \*\*handle, char \*hostname, char \*username, char \*password, char \*using\_clause, char \*process\_name\_version, char \*workflow\_name, double chunk\_size)**

Similar to `connect_local_process()`, this function enables the user to analyze data for pseudo-real-time processing, except that the analysis is connected to the `ares.lbl.gov` server instead of working with a local process HDF5 file.

The first argument “handle” is a pointer to memory internal to the API and may not be altered by the user.

Additional arguments are the “hostname” to connect to, and the “username” and “password” are for authentication. Starting at release 1.2, for security reasons, the “password” is no longer a fixed password. Instead, users need to go to the

<https://ares.lbl.gov> website to obtain a temporary API Key string. The string will be valid for 3 days.

■ **bool getNextChunk(char \*handle)**

The function loads data from the next chunk into memory, either by accessing the HDF5 file in the Local File Mode or the ares.lbl.gov server in the Remote Server Mode. The success or failure of this operation is returned as a Boolean.

The “handle” argument represents the session initiated by either **connect\_\***() functions.

■ **int getNumElem(char \*handle, char \*DS\_name)**

The function returns the number of `DataElements` (data points) in the current chunk for the specified `DataSet` name. If no data from the specified `DataSet` is available, zero is returned.

The first argument “handle” argument represents the session initiated by either **connect\_\***() functions.

The second argument “DS\_name” points to a string containing a `DataSet` name that was specified when initiating the analysis session.

■ **bool getDataElement(char \*handle, char \*DS\_name, int ielem, char \*type, int size, void \*data\_buffer)**

The function copies the specified `DataElement`, for the current chunk, to the user’s memory space, returning the success or failure of this operation as a Boolean. The function checks that the copy operation will not overflow the length of the buffer, as specified by the user.

The first argument “handle” argument represents the session initiated by either **connect\_\***() functions.

The second argument “DS\_name” points to a string containing a `DataSet` name that was specified when initiating the analysis session.

The third argument “ielem” specifies the `DataElement` number in the current chunk.



The fourth argument “type” specifies the data type of the data buffer and can contain the following basic values: “int32”, “uint32”, “float”, “double”, or “char”. The value must match that specified within either the `Data Service` or the HDF5 file.

The fifth argument specifies the size - as in the number of specifically typed elements - of the data buffer allocated by the user.

The sixth argument specifies the memory address of the data buffer already allocated by the user.

■ **stage “family” (char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, ...)**

These functions prepare users’ results prior to actual transmission or writing to a local output file. In this way, multiple results can be returned for each `Chunk`. The success or failure of the operation will be returned via a Boolean. Note that this function will only operate successfully when the session has been initialized with either the **connect\_local\_process()** or **connect\_remote()** methods.

Users must use the appropriate member of the stage family, or the library will not stage the requested datum successfully. The member needed depends on the description of the `DataSet` being staged and the linkage assigned when registered in the `Data Service`.

The following describes the first six arguments, which are common in all cases. The first argument “handle” represents the session initiated by the two aforementioned **connect\_\*()** functions. The second argument “DS\_name” points to a string containing a `DataSet` name of a process output. The third argument “ielem” specifies the `DataElement` number being prepared for recording. The fourth argument “type” specifies the data type of the data buffer and can contain the following basic values: “int32”, “uint32”, “float”, “double”, or “char”. The value must match that specified within either the `Data Service`. The fifth argument specifies the size - as in the number of specifically typed elements - of the data buffer allocated by the user. The sixth argument specifies the memory address of the data buffer already allocated by the user.

The bulleted list below specifies the subroutines, which are members of the stage family. The purpose of each entry is documented along with the C-language signature. Common arguments that sometimes appear in the signature are detailed in this paragraph to provide a more concise reference. The argument “time\_stamp\_value” references the memory address containing the time value of the staged “data\_buffer” datum. The

arguments “xtype” and “x\_buffer” represent pointers to the data type string and a value of x-dimension linked datasets, respectively. The arguments “ytype” and “y\_buffer” represent pointers to the data type string and a value of y-dimension linked datasets, respectively. The arguments “ztype” and “z\_buffer” represent pointers to the data type string and a value of z-dimension linked datasets, respectively. Each of the three individual dimension type and value pairs always appear together, if such a linkage is being made.

- **bool stageData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer)**

Used to write data that is neither time- nor spatially-synced

- **bool stageTimeStampedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value)**

Used to write data that is time- but not spatially-synced

- **bool stageTimeStampedXSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value, char \*xtype, void \*x\_buffer)**

Used to write data that is time- and spatially-synced in the x-dim only

- **bool stageTimeStampedYSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value, char \*ytype, void \*y\_buffer)**

Used to write data that is time- and spatially-synced in the y-dim only

- **bool stageTimeStampedZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value, char \*ztype, void \*z\_buffer)**

Used to write data that is time- and spatially-synced in the z-dim only

- **bool stageTimeStampedXYSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value, char \*xtype, void \*x\_buffer, char \*ytype, void \*y\_buffer)**

Used to write data that is time- and spatially-synced in the x- & y-dims only

- **bool stageTimeStampedXZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value, char \*xtype, void \*x\_buffer, char \*ztype, void \*z\_buffer)**

Used to write data that is time- and spatially-synced in the x- & z-dims only

- **bool stageTimeStampedYZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value, char \*ytype, void \*y\_buffer, char \*ztype, void \*z\_buffer)**

Used to write data that is time- and spatially-synced in the y- & z-dims only

- **bool stageTimeStampedXYZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, double \*time\_stamp\_value, char \*xtype, void \*x\_buffer, char \*ytype, void \*y\_buffer, char \*ztype, void \*z\_buffer)**

Used to write data that is time- and spatially-synced in all 3 dims

- **bool stageXYZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, char \*xtype, void \*x\_buffer, char \*ytype, void \*y\_buffer, char \*ztype, void \*z\_buffer)**

Used to write data that is *NOT* time-synced but is spatially-synced in all 3 dims

- **bool stageXYSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, char \*xtype, void \*x\_buffer, char \*ytype, void \*y\_buffer)**

Used to write data that is *NOT* time-synced but is spatially-synced in only the x- & y-dims

- **bool stageXZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, char \*xtype, void \*x\_buffer, char \*ztype, void \*z\_buffer)**

Used to write data that is *NOT* time-synced but is spatially-synced in only the x- & z-dims

- **bool stageYZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, char \*ytype, void \*y\_buffer, char \*ztype, void \*z\_buffer)**

Used to write data that is *NOT* time-synced but is spatially-synced in only the y- & z-dims

- **bool stageXSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, char \*xtype, void \*x\_buffer)**

Used to write data that is *NOT* time-synced but is spatially-synced in only the x-dim

- **bool stageYSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, char \*ytype, void \*y\_buffer)**

Used to write data that is *NOT* time-synced but is spatially-synced in only the y-dim

- **bool stageZSpatiallySyncedData(char \*handle, char \*DS\_name, unsigned int ielem, char \*type, int size, void \*data\_buffer, char \*ztype, void \*z\_buffer)**

Used to write data that is *NOT* time-synced but is spatially-synced in only the z-dim

- **bool publish(char \*handle)**

In Remote Mode, expect this function to transmit the results prepared using the set of **stage\*()** subroutines to the ares.lbl.gov server and enable access to the next chunk of data. In Local Mode, this function increments an internal counter referencing the current chunk and if the session has been initiated using the **connect\_local\_process()** subroutine, data are written to the output file. *In either case, if this function is not called the user should expect each chunk to contain the same data.*

Starting release-1.2.2, it is required for the users to stage some data before calling **publish()**.

#### ■ **int disconnect(char \*handle)**

The function closes the session represented by the input argument “handle”. It should be called prior to analysis termination to avoid memory leaks. The function returns zero on success, and the user should set the handle to NULL prior to reuse in either connect subroutines.

#### ■ **void setTerminalVerbosity(char \*handle, int level)**

The function sets the terminal output verbosity using the specified verbosity level.

The first argument “handle” argument represents the session initiated by either **connect\_\*()** functions.

The second argument “level” represents the message printing verbosity, which has values 0, 1, 2, and 3 for Error, Warning, Debug, and Info, respectively.

#### ■ **void writeToLogFile(char \*handle, int level, char \*filename)**

The function opens a log file and writes messages using the specified verbosity level.

The first argument “handle” argument represents the session initiated by either **connect\_\*()** functions.

The second argument “level” represents the log message verbosity, which has values 0, 1, 2, and 3 for Error, Warning, Debug and Info, respectively.

The third argument “filename” specifies the full pathname of the log file to be used.

## Installation Instructions

---

Binary libraries are available for the following operating systems: Scientific Linux 6 (RedHat), Macintosh OS X (10.7), Windows Server. Compressed archives for the listed operating systems are available for download from the [ares.lbl.gov](http://ares.lbl.gov) server. After downloading the appropriate archive for your system follow corresponding instructions below.

### POSIX Operating systems (Linux and Mac):

1. Untar the archive: “tar -xf bdcclient.1.2.2-(uname).x86\_64.tar”.
2. Copy the unpacked folder to your desired location. Note: We do not recommend integration of the lib and include directories with the system libraries, directly. Updates will be offered on a semi-regular basis.
3. Source the “set\_env.sh” script at the start of each bash session, by adding a line to your user profile (~/.bashrc).
4. Browse the README in the root directory of the distribution.
5. Browse example code provided in the /bin and /share/examples directories for your intended analysis language.

Note: Example HDF5 input files are located in the /share/data directory.

### Windows Server:

1. Instructions will be added when the binaries become available.

## Wrappers for the Programmatic API

---

Three wrappers are implemented for this programmatic API in Python, Java and Matlab. Each of the wrappers has full implementation of the entire library of subroutines. Instructions on how to use the wrapper are given below. The installation tar ball also contains working example programs or scripts demonstrating any syntax differences and specific usage.

### Python:

1. The python module 'numpy' is required to use this python wrapper.
2. Ensure that you've added both the C-Language Library's location to your system library environment variable and the location of the python wrapper library to your PYTHON\_PATH environment variable either manually or by 'source'-ing the set\_env.sh script provided in this distribution.
3. Browse the README in /share/examples/python for instructions on running the examples.

### Java:

1. Ensure that you've added the C-Language Library's location to your system library environment variable either manually or by 'source'-ing the set\_env.sh script provided in this distribution.
2. The Java wrapper is distributed in the form of a machine independent \*.jar file, that wraps the native C-Library with the Java native interface. The wrapper is distributed in the /share/java directory.
3. Browse the README in /share/examples/java for instructions on compiling and running the examples.

### Matlab:

1. Ensure that you've added the C-Language Library's location to your system library environment variable either manually or by 'source'-ing the set\_env.sh script provided in this distribution.

2. The Matlab wrapper is distributed in the form of a compilable mex source. The wrapper is distributed in the /share/matlab directory. Compilation instructions are provided in the README file in this directory.

Note: On OSX, dynamic libraries end in “dylib” instead of “so”. On Windows the ending is “dll”. Modify the instructions accordingly!

3. Browse the README in /share/examples/matlab for instructions on running the example scripts.



## Quick Reference - API Functions

The table below provides a quick reference to the functions and their purpose in the API.

Function:	Purpose:
<b>connect_local</b>	Initiate a session pulling data from a local HDF5 File.
<b>connect_local_process</b>	Initiate a session pulling data from a local process-tagged HDF5 File.
<b>connect_remote</b>	Initiate a session pulling data from the remote data server.
<b>getNextChunk</b>	Request data from the next pseudo-real-time chunk.
<b>getNumElem</b>	Return the number of data elements for the specified DataSet in the current chunk.
<b>getDataElement</b>	Populate the data buffer with the requested data from the specified DataSet in the current chunk.
<b>stageData</b>	Prepare unlinked results for writing or transmission to the remote data server.
<b>stageTimeStampedData</b>	Prepare results linked in time but not spatial dimensions for writing or transmission to the remote data server.
<b>stageTimeStampedXSpatiallySyncedData</b>	Prepare results linked in time and in the x-dim ( <i>only</i> ) for writing or transmission to the remote data server.
<b>stageTimeStampedYSpatiallySyncedData</b>	Prepare results linked in time and in the y-dim ( <i>only</i> ) for writing or transmission to the remote data server.
<b>stageTimeStampedZSpatiallySyncedData</b>	Prepare results linked in time and in the z-dim ( <i>only</i> ) for writing or transmission to the remote data server.
<b>stageTimeStampedXYSpatiallySyncedData</b>	Prepare results linked in time and in the both x-dim and y-dim for writing or transmission to the remote data server.
<b>stageTimeStampedXZSpatiallySyncedData</b>	Prepare results linked in time and in the both x-dim and z-dim for writing or transmission to the remote data server.

<b>stageTimeStampedYZSpatiallySyncedData</b>	Prepare results linked in time and in the both y-dim and z-dim for writing or transmission to the remote data server.
<b>stageTimeStampedXYZSpatiallySyncedData</b>	Prepare results linked in time and all spatial dimensions for writing or transmission to the remote data server.
<b>stageXYZSpatiallySyncedData</b>	Prepare results linked in ( <i>only</i> ) all spatial dimensions for writing or transmission to the remote data server.
<b>stageXYSpatiallySyncedData</b>	Prepare results linked in both the x-dim and y-dim dimensions for writing or transmission to the remote data server.
<b>stageXZSpatiallySyncedData</b>	Prepare results linked in both the x-dim and z-dim dimensions for writing or transmission to the remote data server.
<b>stageYZSpatiallySyncedData</b>	Prepare results linked in both the y-dim and z-dim dimensions for writing or transmission to the remote data server.
<b>stageXSpatiallySyncedData</b>	Prepare results linked in ( <i>only</i> ) the x-dim for writing or transmission to the remote data server.
<b>stageYSpatiallySyncedData</b>	Prepare results linked in ( <i>only</i> ) the y-dim for writing or transmission to the remote data server.
<b>stageZSpatiallySyncedData</b>	Prepare results linked in ( <i>only</i> ) the z-dim for writing or transmission to the remote data server.
<b>publish</b>	Transmit results to the remote server if in Remote Mode and advance the chunk counter.
<b>disconnect</b>	Close a session and free memory allocated by the library.
<b>setTerminalVerbosity</b>	Set the message printing verbosity for terminal output.
<b>writeToLogFile</b>	Set the message logging verbosity and open the log file for writing.